RTEMS C User's Guide

Edition 4.0.0, for RTEMS 4.0.0

October 1998

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 1.1.1.1

Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to either:

```
On-Line Applications Research Corporation
4910-L Corporate Drive
Huntsville, AL 35805
VOICE: (256) 722-9985
FAX:   (256) 722-0985
EMAIL: rtems@OARcorp.com
```

Preface

In recent years, the cost required to develop a software product has increased significantly while the target hardware costs have decreased. Now a larger portion of money is expended in developing, using, and maintaining software. The trend in computing costs is the complete dominance of software over hardware costs. Because of this, it is necessary that formal disciplines be established to increase the probability that software is characterized by a high degree of correctness, maintainability, and portability. In addition, these disciplines must promote practices that aid in the consistent and orderly development of a software system within schedule and budgetary constraints. To be effective, these disciplines must adopt standards which channel individual software efforts toward a common goal.

The push for standards in the software development field has been met with various degrees of success. The Microprocessor Operating Systems Interfaces (MOSI) effort has experienced only limited success. As popular as the UNIX operating system has grown, the attempt to develop a standard interface definition to allow portable application development has only recently begun to produce the results needed in this area. Unfortunately, very little effort has been expended to provide standards addressing the needs of the real-time community. Several organizations have addressed this need during recent years.

The Real Time Executive Interface Definition (RTEID) was developed by Motorola with technical input from Software Components Group. RTEID was adopted by the VMEbus International Trade Association (VITA) as a baseline draft for their proposed standard multiprocessor, real-time executive interface, Open Real-Time Kernel Interface Definition (ORKID). These two groups are currently working together with the IEEE P1003.4 committee to insure that the functionality of their proposed standards is adopted as the real-time extensions to POSIX.

This emerging standard defines an interface for the development of real-time software to ease the writing of real-time application programs that are directly portable across multiple real-time executive implementations. This interface includes both the source code interfaces and run-time behavior as seen by a real-time application. It does not include the details of how a kernel implements these functions. The standard's goal is to serve as a complete definition of external interfaces so that application code that conforms to these interfaces will execute properly in all real-time executive environments. With the use of a standards compliant executive, routines that acquire memory blocks, create and manage message queues, establish and use semaphores, and send and receive signals need not be redeveloped for a different real-time environment as long as the new environment is compliant with the standard. Software developers need only concentrate on the hardware dependencies of the real-time system. Furthermore, most hardware dependencies for real-time applications can be localized to the device drivers.

A compliant executive provides simple and flexible real-time multiprocessing. It easily lends itself to both tightly-coupled and loosely-coupled configurations (depending on the system hardware configuration). Objects such as tasks, queues, events, signals, semaphores, and memory blocks can be designated as global objects and accessed by any task regardless of which processor the object and the accessing task reside.

The acceptance of a standard for real-time executives will produce the same advantages enjoyed from the push for UNIX standardization by AT&T's System V Interface Definition and IEEE's POSIX efforts. A compliant multiprocessing executive will allow close coupling between UNIX systems and real-time executives to provide the many benefits of the UNIX development environment to be applied to real-time software development. Together they provide the necessary laboratory environment to implement real-time, distributed, embedded systems using a wide variety of computer architectures.

A study was completed in 1988, within the Research, Development, and Engineering Center, U.S. Army Missile Command, which compared the various aspects of the Ada programming language as they related to the application of Ada code in distributed and/or multiple processing systems. Several critical conclusions were derived from the study. These conclusions have a major impact on the way the Army develops application software for embedded applications. These impacts apply to both in-house software development and contractor developed software.

A conclusion of the analysis, which has been previously recognized by other agencies attempting to utilize Ada in a distributed or multiprocessing environment, is that the Ada programming language does not adequately support multiprocessing. Ada does provide a mechanism for multi-tasking, however, this capability exists only for a single processor system. The language also does not have inherent capabilities to access global named variables, flags or program code. These critical features are essential in order for data to be shared between processors. However, these drawbacks do have workarounds which are sometimes awkward and defeat the intent of software maintainability and portability goals.

Another conclusion drawn from the analysis, was that the run time executives being delivered with the Ada compilers were too slow and inefficient to be used in modern missile systems. A run time executive is the core part of the run time system code, or operating system code, that controls task scheduling, input/output management and memory management. Traditionally, whenever efficient executive (also known as kernel) code was required by the application, the user developed in-house software. This software was usually written in assembly language for optimization.

Because of this shortcoming in the Ada programming language, software developers in research and development and contractors for project managed systems, are mandated by technology to purchase and utilize off-the-shelf third party kernel code. The contractor, and eventually the Government, must pay a licensing fee for every copy of the kernel code used in an embedded system.

The main drawback to this development environment is that the Government does not own, nor has the right to modify code contained within the kernel. V&V techniques in this situation are more difficult than if the complete source code were available. Responsibility for system failures due to faulty software is yet another area to be resolved under this environment.

The Guidance and Control Directorate began a software development effort to address these problems. A project to develop an experimental run time kernel was begun that will eliminate the major drawbacks of the Ada programming language mentioned above. The Real Time Executive for Multiprocessor Systems (RTEMS) provides full capabilities for management of tasks, interrupts, time, and multiple processors in addition to those features typical of generic operating systems. The code is Government owned, so no licensing fees are necessary. RTEMS has been implemented in both the Ada and C programming languages. It has been ported to the following processor families:

- Intel i80386 and above
- Intel i80960
- Motorola MC68xxx
- Motorola MC683xx
- MIPS
- PowerPC
- SPARC
- Hewlett Packard PA-RISC
- Hitach SH
- AMD A29K
- UNIX

Support for other processor families, including RISC, CISC, and DSP, is planned. Since almost all of RTEMS is written in a high level language, ports to additional processor families require minimal effort.

RTEMS multiprocessor support is capable of handling either homogeneous or heterogeneous systems. The kernel automatically compensates for architectural differences (byte swapping, etc.) between processors. This allows a much easier transition from one processor family to another without a major system redesign.

Since the proposed standards are still in draft form, RTEMS cannot and does not claim compliance. However, the status of the standard is being carefully monitored to guarantee that RTEMS provides the functionality specified in the standard. Once approved, RTEMS will be made compliant.

This document is a detailed users guide for a functionally compliant real-time multiprocessor executive. It describes the user interface and run-time behavior of Release {No value for “RELEASE”} of the C interface to RTEMS.

1 Overview

1.1 Introduction

RTEMS, Real-Time Executive for Multiprocessor Systems, is a real-time executive (kernel) which provides a high performance environment for embedded military applications including the following features:

- multitasking capabilities
- homogeneous and heterogeneous multiprocessor systems
- event-driven, priority-based, preemptive scheduling
- optional rate monotonic scheduling
- intertask communication and synchronization
- priority inheritance
- responsive interrupt management
- dynamic memory allocation
- high level of user configurability

This manual describes the usage of RTEMS for applications written in the C programming language. Those implementation details that are processor dependent are provided in the Applications Supplement documents. A supplement document which addresses specific architectural issues that affect RTEMS is provided for each processor type that is supported.

1.2 Real-time Application Systems

Real-time application systems are a special class of computer applications. They have a complex set of characteristics that distinguish them from other software problems. Generally, they must adhere to more rigorous requirements. The correctness of the system depends not only on the results of computations, but also on the time at which the results are produced. The most important and complex characteristic of real-time application systems is that they must receive and respond to a set of external stimuli within rigid and critical time constraints referred to as deadlines. Systems can be buried by an avalanche of interdependent, asynchronous or cyclical event streams.

Deadlines can be further characterized as either hard or soft based upon the value of the results when produced after the deadline has passed. A deadline is hard if the results have no value or if their use will result in a catastrophic event. In contrast, results which are produced after a soft deadline may have some value.

Another distinguishing requirement of real-time application systems is the ability to coordinate or manage a large number of concurrent activities. Since software is a synchronous entity, this presents special problems. One instruction follows another in a repeating synchronous cycle. Even though mechanisms have been developed to allow for the processing of external asynchronous events, the software design efforts required to process and manage these events and tasks are growing more complicated.

The design process is complicated further by spreading this activity over a set of processors instead of a single processor. The challenges associated with designing and building real-time application systems become very complex when multiple processors are involved. New requirements such as interprocessor communication channels and global resources that must be shared between competing processors are introduced. The ramifications of multiple processors complicate each and every characteristic of a real-time system.

1.3 Real-time Executive

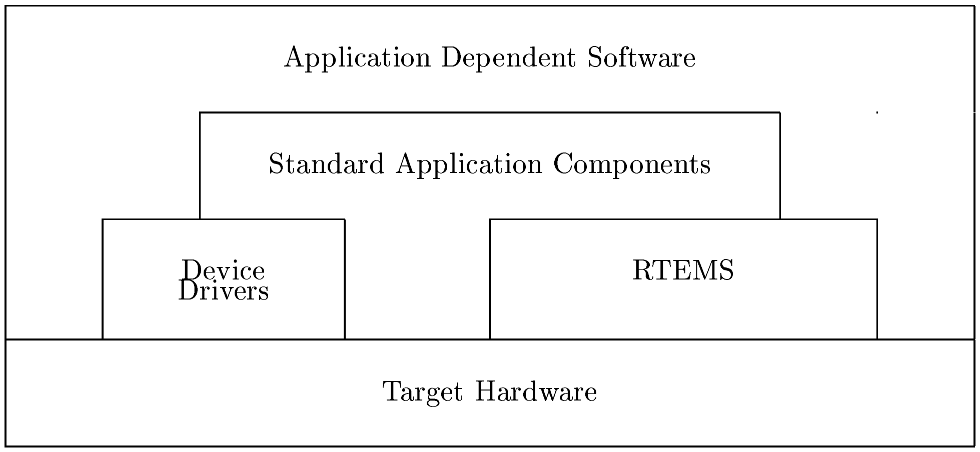
Fortunately, real-time operating systems or real-time executives serve as a cornerstone on which to build the application system. A real-time multitasking executive allows an application to be cast into a set of logical, autonomous processes or tasks which become quite manageable. Each task is internally synchronous, but different tasks execute independently, resulting in an asynchronous processing stream. Tasks can be dynamically paused for many reasons resulting in a different task being allowed to execute for a period of time. The executive also provides an interface to other system components such as interrupt handlers and device drivers. System components may request the executive to allocate and coordinate resources, and to wait for and trigger synchronizing conditions. The executive system calls effectively extend the CPU instruction set to support efficient multitasking. By causing tasks to travel through well-defined state transitions, system calls permit an application to demand-switch between tasks in response to real-time events.

By proper grouping of responses to stimuli into separate tasks, a system can now asynchronously switch between independent streams of execution, directly responding to external stimuli as they occur. This allows the system design to meet critical performance specifications which are typically measured by guaranteed response time and transaction throughput. The multiprocessor extensions of RTEMS provide the features necessary to manage the extra requirements introduced by a system distributed across several processors. It removes the physical barriers of processor boundaries from the world of the system designer, enabling more critical aspects of the system to receive the required attention. Such a system, based on an efficient real-time, multiprocessor executive, is a more realistic model of the outside world or environment for which it is designed. As a result, the system will always be more logical, efficient, and reliable.

By using the directives provided by RTEMS, the real-time applications developer is freed from the problem of controlling and synchronizing multiple tasks and processors. In addition, one need not develop, test, debug, and document routines to manage memory, pass messages, or provide mutual exclusion. The developer is then able to concentrate solely on the application. By using standard software components, the time and cost required to develop sophisticated real-time applications is significantly reduced.

1.4 RTEMS Application Architecture

One important design goal of RTEMS was to provide a bridge between two critical layers of typical real-time systems. As shown in the following figure, RTEMS serves as a buffer between the project dependent application code and the target hardware. Most hardware dependencies for real-time applications can be localized to the low level device drivers. The RTEMS I/O interface manager provides an efficient tool for incorporating these hardware dependencies into the system while simultaneously providing a general mechanism to the application code that accesses them. A well designed real-time system can benefit from this architecture by building a rich library of standard application components which can be used repeatedly in other real-time projects.


Application Dependent Software
Standard Application Components
Device Drivers
RTEMS
Target Hardware

1.5 RTEMS Internal Architecture

RTEMS can be viewed as a set of layered components that work in harmony to provide a set of services to a real-time application system. The executive interface presented to the application is formed by grouping directives into logical sets called resource managers. Functions utilized by multiple managers such as scheduling, dispatching, and object management are provided in the executive core. The executive core depends on a small set of CPU dependent routines. Together these components provide a powerful run time environment that promotes the development of efficient real-time application systems. The following figure illustrates this organization:

```
+-----------------------------------------------+
|          RTEMS Executive Interface            |
+-----------------------------------------------+
|                 RTEMS Core                    |
+-----------------------------------------------+
|              CPU Dependent Code               |
+-----------------------------------------------+
```

Subsequent chapters present a detailed description of the capabilities provided by each of the following RTEMS managers:

- initialization
- task
- interrupt
- clock
- timer
- semaphore
- message
- event
- signal
- partition
- region
- dual ported memory
- I/O
- fatal error
- rate monotonic
- user extensions
- multiprocessing

1.6 User Customization and Extensibility

As thirty-two bit microprocessors have decreased in cost, they have become increasingly common in a variety of embedded systems. A wide range of custom and general-purpose processor boards are based on various thirty-two bit processors. RTEMS was designed to make no assumptions concerning the characteristics of individual microprocessor families or of specific support hardware. In addition, RTEMS allows the system developer a high degree of freedom in customizing and extending its features.

RTEMS assumes the existence of a supported microprocessor and sufficient memory for both RTEMS and the real-time application. Board dependent components such as clocks, interrupt controllers, or I/O devices can be easily integrated with RTEMS. The customization and extensibility features allow RTEMS to efficiently support as many environments as possible.

1.7 Portability

The issue of portability was the major factor in the creation of RTEMS. Since RTEMS is designed to isolate the hardware dependencies in the specific board support packages, the real-time application should be easily ported to any other processor. The use of RTEMS allows the development of real-time applications which can be completely independent of a particular microprocessor architecture.

1.8 Memory Requirements

Since memory is a critical resource in many real-time embedded systems, RTEMS was specifically designed to allow unused managers to be excluded from the run-time environment. This allows the application designer the flexibility to tailor RTEMS to most efficiently meet system requirements while still satisfying even the most stringent memory constraints. As a result, the size of the RTEMS executive is application dependent. A worksheet is provided in the Memory Requirements chapter of the Applications Supplement document for a specific target processor. The worksheet is used to calculate the memory requirements of a custom RTEMS run-time environment. The following managers may be optionally excluded:

- clock
- timer
- semaphore
- message
- event
- signal
- partition
- region
- dual ported memory
- I/O
- rate monotonic
- fatal error
- multiprocessing

RTEMS utilizes memory for both code and data space. Although RTEMS' data space must be in RAM, its code space can be located in either ROM or RAM.

1.9 Audience

This manual was written for experienced real-time software developers. Although some background is provided, it is assumed that the reader is familiar with the concepts of task management as well as intertask communication and synchronization. Since directives, user related data structures, and examples are presented in C, a basic understanding of the C programming language is required to fully understand the material presented. However, because of the similarity of the Ada and C RTEMS implementations, users will find that the use and behavior of the two implementations is very similar. A working knowledge of the target processor is helpful in understanding some of RTEMS' features. A thorough understanding of the executive cannot be obtained without studying the entire manual because many of RTEMS' concepts and features are interrelated. Experienced RTEMS users will find that the manual organization facilitates its use as a reference document.

1.10 Conventions

The following conventions are used in this manual:

- Significant words or phrases as well as all directive names are printed in bold type.
- Items in bold capital letters are constants defined by RTEMS. Each language interface provided by RTEMS includes a file containing the standard set of constants, data types, and structure definitions which can be incorporated into the user application.
- A number of type definitions are provided by RTEMS and can be found in rtems.h.
- The characters "0x" preceding a number indicates that the number is in hexadecimal format. Any other numbers are assumed to be in decimal format.

1.11 Manual Organization

This first chapter has presented the introductory and background material for the RTEMS executive. The remaining chapters of this manual present a detailed description of RTEMS and the environment, including run time behavior, it creates for the user.

A chapter is dedicated to each manager and provides a detailed discussion of each RTEMS manager and the directives which it provides. The presentation format for each directive includes the following sections:

- Calling sequence
- Directive status codes
- Description
- Notes

The following provides an overview of the remainder of this manual:

Chapter 2	Key Concepts: presents an introduction to the ideas which are common across multiple RTEMS managers.
Chapter 3:	Initialization Manager: describes the functionality and directives provided by the Initialization Manager.
Chapter 4:	Task Manager: describes the functionality and directives provided by the Task Manager.
Chapter 5:	Interrupt Manager: describes the functionality and directives provided by the Interrupt Manager.
Chapter 6:	Clock Manager: describes the functionality and directives provided by the Clock Manager.
Chapter 7	Timer Manager: describes the functionality and directives provided by the Timer Manager.

Chapter 8:	Semaphore Manager: describes the functionality and directives provided by the Semaphore Manager.
Chapter 9:	Message Manager: describes the functionality and directives provided by the Message Manager.
Chapter 10:	Event Manager: describes the functionality and directives provided by the Event Manager.
Chapter 11:	Signal Manager: describes the functionality and directives provided by the Signal Manager.
Chapter 12:	Partition Manager: describes the functionality and directives provided by the Partition Manager.
Chapter 13:	Region Manager: describes the functionality and directives provided by the Region Manager.
Chapter 14:	Dual-Ported Memory Manager: describes the functionality and directives provided by the Dual-Ported Memory Manager.
Chapter 15:	I/O Manager: describes the functionality and directives provided by the I/O Manager.
Chapter 16:	Fatal Error Manager: describes the functionality and directives provided by the Fatal Error Manager.
Chapter 17:	Scheduling Concepts: details the RTEMS scheduling algorithm and task state transitions.
Chapter 18:	Rate Monotonic Manager: describes the functionality and directives provided by the Rate Monotonic Manager.
Chapter 19:	Board Support Packages: defines the functionality required of user-supplied board support packages.
Chapter 20:	User Extensions: shows the user how to extend RTEMS to incorporate custom features.
Chapter 21:	Configuring a System: details the process by which one tailors RTEMS for a particular single-processor or multiprocessor application.
Chapter 22:	Multiprocessing Manager: presents a conceptual overview of the multiprocessing capabilities provided by RTEMS as well as describing the Multiprocessing Communications Interface Layer and Multiprocessing Manager directives.
Chapter 23:	Directive Status Codes: provides a definition of each of the directive status codes referenced in this manual.
Chapter 24:	Example Application: provides a template for simple RTEMS applications.
Chapter 25:	Glossary: defines terms used throughout this manual.

2 Key Concepts

2.1 Introduction

The facilities provided by RTEMS are built upon a foundation of very powerful concepts. These concepts must be understood before the application developer can efficiently utilize RTEMS. The purpose of this chapter is to familiarize one with these concepts.

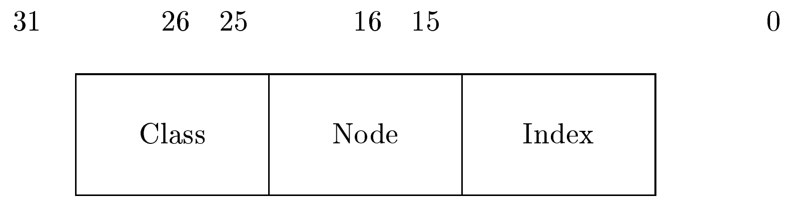
2.2 Objects

RTEMS provides directives which can be used to dynamically create, delete, and manipulate a set of predefined object types. These types include tasks, message queues, semaphores, memory regions, memory partitions, timers, ports, and rate monotonic periods. The object-oriented nature of RTEMS encourages the creation of modular applications built upon re-usable "building block" routines.

All objects are created on the local node as required by the application and have an RTEMS assigned ID. All objects have a user-assigned name. Although a relationship exists between an object's name and its RTEMS assigned ID, the name and ID are not identical. Object names are completely arbitrary and selected by the user as a meaningful "tag" which may commonly reflect the object's use in the application. Conversely, object IDs are designed to facilitate efficient object manipulation by the executive.

An object name is an unsigned thirty-two bit entity associated with the object by the user. Although not required by RTEMS, object names are typically composed of four ASCII characters which help identify that object. For example, a task which causes a light to blink might be called "LITE". Utilities are provided to build an object name from four ASCII characters and to decompose an object name into four ASCII characters. However, it is not required that the application use ASCII characters to build object names. For example, if an application requires one-hundred tasks, it would be difficult to assign meaningful ASCII names to each task. A more convenient approach would be to name them the binary values one through one-hundred, respectively.

An object ID is a unique unsigned thirty-two bit entity composed of three parts: object class, node, and index. The most significant six bits are the object class. The next ten bits are the number of the node on which this object was created. The node number is always one (1) in a single processor system. The least significant sixteen bits form an identifier within a particular object type. This identifier, called the object index, ranges in value from 1 to the maximum number of objects configured for this object type.



The three components of an object ID make it possible to quickly locate any object in even the most complicated multiprocessor system. Object ID's are associated with an object by RTEMS when the object is created and the corresponding ID is returned by the appropriate object create directive. The object ID is required as input to all directives involving objects, except those which create an object or obtain the ID of an object.

The object identification directives can be used to dynamically obtain a particular object's ID given its name. This mapping is accomplished by searching the name table associated with this object type. If the name is non-unique, then the ID associated with the first occurrence of the name will be returned to the application. Since object IDs are returned when the object is created, the object identification directives are not necessary in a properly designed single processor application.

An object control block is a data structure defined by RTEMS which contains the information necessary to manage a particular object type. For efficiency reasons, the format of each object type's control block is different. However, many of the fields are similar in function. The number of each type of control block is application dependent and determined by the values specified in the user's Configuration Table. An object control block is allocated at object create time and freed when the object is deleted. With the exception of user extension routines, object control blocks are not directly manipulated by user applications.

2.3 Communication and Synchronization

In real-time multitasking applications, the ability for cooperating execution threads to communicate and synchronize with each other is imperative. A real-time executive should provide an application with the following capabilities:

- Data transfer between cooperating tasks
- Data transfer between tasks and ISRs
- Synchronization of cooperating tasks

- Synchronization of tasks and ISRs

Most RTEMS managers can be used to provide some form of communication and/or synchronization. However, managers dedicated specifically to communication and synchronization provide well established mechanisms which directly map to the application's varying needs. This level of flexibility allows the application designer to match the features of a particular manager with the complexity of communication and synchronization required. The following managers were specifically designed for communication and synchronization:

- Semaphore
- Message Queue
- Event
- Signal

The semaphore manager supports mutual exclusion involving the synchronization of access to one or more shared user resources. Binary semaphores may utilize the optional priority inheritance algorithm to avoid the problem of priority inversion. The message manager supports both communication and synchronization, while the event manager primarily provides a high performance synchronization mechanism. The signal manager supports only asynchronous communication and is typically used for exception handling.

2.4 Time

The development of responsive real-time applications requires an understanding of how RTEMS maintains and supports time-related operations. The basic unit of time in RTEMS is known as a tick. The frequency of clock ticks is completely application dependent and determines the granularity and accuracy of all interval and calendar time operations.

By tracking time in units of ticks, RTEMS is capable of supporting interval timing functions such as task delays, timeouts, timeslicing, the delayed execution of timer service routines, and the rate monotonic scheduling of tasks. An interval is defined as a number of ticks relative to the current time. For example, when a task delays for an interval of ten ticks, it is implied that the task will not execute until ten clock ticks have occurred.

A characteristic of interval timing is that the actual interval period may be a fraction of a tick less than the interval requested. This occurs because the time at which the delay timer is set up occurs at some time between two clock ticks. Therefore, the first countdown tick occurs in less than the complete time interval for a tick. This can be a problem if the clock granularity is large.

The rate monotonic scheduling algorithm is a hard real-time scheduling methodology. This methodology provides rules which allows one to guarantee that a set of independent periodic tasks will always meet their deadlines – even under transient overload conditions. The rate monotonic manager provides directives built upon the Clock Manager's interval timer support routines.

Interval timing is not sufficient for the many applications which require that time be kept in wall time or true calendar form. Consequently, RTEMS maintains the current date and time. This allows selected time operations to be scheduled at an actual calendar date and time. For example, a task could request to delay until midnight on New Year's Eve before lowering the ball at Times Square.

Obviously, the directives which use intervals or wall time cannot operate without some external mechanism which provides a periodic clock tick. This clock tick is typically provided by a real time clock or counter/timer device.

2.5 Memory Management

RTEMS memory management facilities can be grouped into two classes: dynamic memory allocation and address translation. Dynamic memory allocation is required by applications whose memory requirements vary through the application's course of execution. Address translation is needed by applications which share memory with another CPU or an intelligent Input/Output processor. The following RTEMS managers provide facilities to manage memory:

- Region
- Partition
- Dual Ported Memory

RTEMS memory management features allow an application to create simple memory pools of fixed size buffers and/or more complex memory pools of variable size segments. The partition manager provides directives to manage and maintain pools of fixed size entities such as resource control blocks. Alternatively, the region manager provides a more general purpose memory allocation scheme that supports variable size blocks of memory which are dynamically obtained and freed by the application. The dual-ported memory manager provides executive support for address translation between internal and external dual-ported RAM address space.

3 Initialization Manager

3.1 Introduction

The initialization manager is responsible for initiating and shutting down RTEMS. Initiating RTEMS involves creating and starting all configured initialization tasks, and for invoking the initialization routine for each user-supplied device driver. In a multiprocessor configuration, this manager also initializes the interprocessor communications layer. The directives provided by the initialization manager are:

- `rtems_initialize_executive` - Initialize RTEMS
- `rtems_initialize_executive_early` - Initialize RTEMS and do NOT Start Multitasking
- `rtems_initialize_executive_late` - Complete Initialization and Start Multitasking
- `rtems_shutdown_executive` - Shutdown RTEMS

3.2 Background

3.2.1 Initialization Tasks

Initialization task(s) are the mechanism by which RTEMS transfers initial control to the user's application. Initialization tasks differ from other application tasks in that they are defined in the User Initialization Tasks Table and automatically created and started by RTEMS as part of its initialization sequence. Since the initialization tasks are scheduled using the same algorithm as all other RTEMS tasks, they must be configured at a priority and mode which will insure that they will complete execution before other application tasks execute. Although there is no upper limit on the number of initialization tasks, an application is required to define at least one.

A typical initialization task will create and start the static set of application tasks. It may also create any other objects used by the application. Initialization tasks which only perform initialization should delete themselves upon completion to free resources for other tasks. Initialization tasks may transform themselves into a "normal" application task. This transformation typically involves changing priority and execution mode. RTEMS does not automatically delete the initialization tasks.

3.2.2 The System Initialization Task

The System Initialization Task is responsible for initializing all device drivers. As a result, this task has a higher priority than all other tasks to insure that no application tasks executes until all device drivers are initialized. After device initialization in a single processor system, this task will delete itself.

The System Initialization Task must have enough stack space to successfully execute the initialization routines for all device drivers and, in multiprocessor configurations, the Multiprocessor Communications Interface Layer initialization routine. The CPU Configuration Table contains a field which allows the application or BSP to increase the default amount of stack space allocated for this task.

In multiprocessor configurations, the System Initialization Task does not delete itself after initializing the device drivers. Instead it transforms itself into the Multiprocessing Server which initializes the Multiprocessor Communications Interface Layer, verifies multiprocessor system consistency, and processes all requests from remote nodes.

3.2.3 The Idle Task

The Idle Task is the lowest priority task in a system and executes only when no other task is ready to execute. This task consists of an infinite loop and will be preempted when any other task is made ready to execute.

3.2.4 Initialization Manager Failure

The fatal_error_occurred directive will be called from `rtems_initialize_executive` for any of the following reasons:

- If either the Configuration Table or the CPU Dependent Information Table is not provided.
- If the starting address of the RTEMS RAM Workspace, supplied by the application in the Configuration Table, is NULL or is not aligned on a four-byte boundary.
- If the size of the RTEMS RAM Workspace is not large enough to initialize and configure the system.
- If the interrupt stack size specified is too small.
- If multiprocessing is configured and the node entry in the Multiprocessor Configuration Table is not between one and the maximum_nodes entry.
- If a multiprocessor system is being configured and no Multiprocessor Communications Interface is specified.
- If no user initialization tasks are configured. At least one initialization task must be configured to allow RTEMS to pass control to the application at the end of the executive initialization sequence.
- If any of the user initialization tasks cannot be created or started successfully.

3.3 Operations

3.3.1 Initializing RTEMS

The `rtems_initialize_executive` directive is called by the board support package at the completion of its initialization sequence. RTEMS assumes that the board support package successfully completed its initialization activities. The `rtems_initialize_executive` directive completes the initialization sequence by performing the following actions:

- Initializing internal RTEMS variables;
- Allocating system resources;
- Creating and starting the System Initialization Task;
- Creating and starting the Idle Task;
- Creating and starting the user initialization task(s); and
- Initiating multitasking.

This directive MUST be called before any other RTEMS directives. The effect of calling any RTEMS directives before `rtems_initialize_executive` is unpredictable. Many of RTEMS actions during initialization are based upon the contents of the Configuration Table and CPU Dependent Information Table. For more information regarding the format and contents of these tables, please refer to the chapter Configuring a System.

The final step in the initialization sequence is the initiation of multitasking. When the scheduler and dispatcher are enabled, the highest priority, ready task will be dispatched to run. Control will not be returned to the board support package after multitasking is enabled until `rtems_shutdown_executive` the directive is called.

The `rtems_initialize_executive` directive provides a conceptually simple way to initialize RTEMS. However, in certain cases, this mechanism cannot be used. The `rtems_initialize_executive_early` and `rtems_initialize_executive_late` directives are provided as an alternative mechanism for initializing RTEMS. The `rtems_initialize_executive_early` directive returns to the caller BEFORE initiating multitasking. The `rtems_initialize_executive_late` directive is invoked to start multitasking. It is critical that only one of the RTEMS initialization sequences be used in an application.

3.3.2 Shutting Down RTEMS

The `rtems_shutdown_executive` directive is invoked by the application to end multitasking and return control to the board support package. The board support package resumes execution at the code immediately following the invocation of the `rtems_initialize_executive` directive.

3.4 Directives

This section details the initialization manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

3.4.1 INITIALIZE_EXECUTIVE - Initialize RTEMS

CALLING SEQUENCE:

```
void rtems_initialize_executive(
  rtems_configuration_table *configuration_table,
  rtems_cpu_table           *cpu_table
);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive is called when the board support package has completed its initialization to allow RTEMS to initialize the application environment based upon the information in the Configuration Table, CPU Dependent Information Table, User Initialization Tasks Table, Device Driver Table, User Extension Table, Multiprocessor Configuration Table, and the Multiprocessor Communications Interface (MPCI) Table. This directive starts multitasking and does not return to the caller until the `rtems_shutdown_executive` directive is invoked.

NOTES:

This directive MUST be the first RTEMS directive called and it DOES NOT RETURN to the caller until the `rtems_shutdown_executive` is invoked.

This directive causes all nodes in the system to verify that certain configuration parameters are the same as those of the local node. If an inconsistency is detected, then a fatal error is generated.

The application must use only one of the two initialization sequences: `rtems_initialize_executive` or `rtems_initialize_executive_early` and `rtems_initialize_executive_late`. The `rtems_initialize_executive` directive is logically equivalent to invoking `rtems_initialize_executive_early` and `rtems_initialize_executive_late` with no intervening actions.

3.4.2 INITIALIZE_EXECUTIVE_EARLY - Initialize RTEMS and do NOT Start Multitasking

CALLING SEQUENCE:

```
rtems_interrupt_level rtems_initialize_executive_early(
  rtems_configuration_table *configuration_table,
  rtems_cpu_table           *cpu_table
);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive is called when the board support package has completed its initialization to allow RTEMS to initialize the application environment based upon the information in the Configuration Table, CPU Dependent Information Table, User Initialization Tasks Table, Device Driver Table, User Extension Table, Multiprocessor Configuration Table, and the Multiprocessor Communications Interface (MPCI) Table. This directive returns to the caller after completing the basic RTEMS initialization but before multitasking is initiated. The interrupt level in place when the directive is invoked is returned to the caller. This interrupt level should be the same one passed to `rtems_initialize_executive_late`.

NOTES:

The application must use only one of the two initialization sequences: `rtems_initialize_executive` or `rtems_nitialize_executive_early` and `rtems_nitialize_executive_late`.

3.4.3 INITIALIZE_EXECUTIVE_LATE - Complete Initialization and Start Multitasking

CALLING SEQUENCE:

```
void rtems_initialize_executive_late(
  rtems_interrupt_level  bsp_level
);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive is called after the `rtems_initialize_executive_early` directive has been called to complete the RTEMS initialization sequence and initiate multitasking. The interrupt level returned by the `rtems_initialize_executive_early` directive should be in bsp_level and this value is restored as part of this directive returning to the caller after the `rtems_shutdown_executive` directive is invoked.

NOTES:

This directive MUST be the second RTEMS directive called and it DOES NOT RETURN to the caller until the `rtems_shutdown_executive` is invoked.

This directive causes all nodes in the system to verify that certain configuration parameters are the same as those of the local node. If an inconsistency is detected, then a fatal error is generated.

The application must use only one of the two initialization sequences: `rtems_initialize_executive` or `rtems_nitialize_executive_early` and `rtems_initialize_executive_late`.

3.4.4 SHUTDOWN_EXECUTIVE - Shutdown RTEMS

CALLING SEQUENCE:

```
void rtems_shutdown_executive(
  rtems_unsigned32 result
);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive is called when the application wishes to shutdown RTEMS and return control to the board support package. The board support package resumes execution at the code immediately following the invocation of the `rtems_initialize_executive` directive.

NOTES:

This directive MUST be the last RTEMS directive invoked by an application and it DOES NOT RETURN to the caller.

This directive should not be invoked until the executive has successfully completed initialization.

4 Task Manager

4.1 Introduction

The task manager provides a comprehensive set of directives to create, delete, and administer tasks. The directives provided by the task manager are:

- `rtems_task_create` - Create a task
- `rtems_task_ident` - Get ID of a task
- `rtems_task_start` - Start a task
- `rtems_task_restart` - Restart a task
- `rtems_task_delete` - Delete a task
- `rtems_task_suspend` - Suspend a task
- `rtems_task_resume` - Resume a task
- `rtems_task_set_priority` - Set task priority
- `rtems_task_mode` - Change current task's mode
- `rtems_task_get_note` - Get task notepad entry
- `rtems_task_set_note` - Set task notepad entry
- `rtems_task_wake_after` - Wake up after interval
- `rtems_task_wake_when` - Wake up when specified

4.2 Background

4.2.1 Task Definition

Many definitions of a task have been proposed in computer literature. Unfortunately, none of these definitions encompasses all facets of the concept in a manner which is operating system independent. Several of the more common definitions are provided to enable each user to select a definition which best matches their own experience and understanding of the task concept:

- a "dispatchable" unit.
- an entity to which the processor is allocated.
- an atomic unit of a real-time, multiprocessor system.
- single threads of execution which concurrently compete for resources.
- a sequence of closely related computations which can execute concurrently with other computational sequences.

From RTEMS' perspective, a task is the smallest thread of execution which can compete on its own for system resources. A task is manifested by the existence of a task control block (TCB).

4.2.2 Task Control Block

The Task Control Block (TCB) is an RTEMS defined data structure which contains all the information that is pertinent to the execution of a task. During system initialization, RTEMS reserves a TCB for each task configured. A TCB is allocated upon creation of the task and is returned to the TCB free list upon deletion of the task.

The TCB's elements are modified as a result of system calls made by the application in response to external and internal stimuli. TCBs are the only RTEMS internal data structure that can be accessed by an application via user extension routines. The TCB contains a task's name, ID, current priority, current and starting states, execution mode, set of notepad locations, TCB user extension pointer, scheduling control structures, as well as data required by a blocked task.

A task's context is stored in the TCB when a task switch occurs. When the task regains control of the processor, its context is restored from the TCB. When a task is restarted, the initial state of the task is restored from the starting context area in the task's TCB.

4.2.3 Task States

A task may exist in one of the following five states:

- **executing** - Currently scheduled to the CPU
- **ready** - May be scheduled to the CPU
- **blocked** - Unable to be scheduled to the CPU
- **dormant** - Created task that is not started
- **non-existent** - Uncreated or deleted task

An active task may occupy the executing, ready, blocked or dormant state, otherwise the task is considered non-existent. One or more tasks may be active in the system simultaneously. Multiple tasks communicate, synchronize, and compete for system resources with each other via system calls. The multiple tasks appear to execute in parallel, but actually each is dispatched to the CPU for periods of time determined by the RTEMS scheduling algorithm. The scheduling of a task is based on its current state and priority.

4.2.4 Task Priority

A task's priority determines its importance in relation to the other tasks executing on the same processor. RTEMS supports 255 levels of priority ranging from 1 to 255. Tasks of numerically smaller priority values are more important tasks than tasks of numerically larger priority values. For example, a task at priority level 5 is of higher privilege than a task at priority level 10. There is no limit to the number of tasks assigned to the same priority.

Each task has a priority associated with it at all times. The initial value of this priority is assigned at task creation time. The priority of a task may be changed at any subsequent time.

Priorities are used by the scheduler to determine which ready task will be allowed to execute. In general, the higher the logical priority of a task, the more likely it is to receive processor execution time.

4.2.5 Task Mode

A task's mode is a combination of the following four components:

- preemption
- ASR processing
- timeslicing
- interrupt level

It is used to modify RTEMS' scheduling process and to alter the execution environment of the task.

The preemption component allows a task to determine when control of the processor is relinquished. If preemption is disabled (`RTEMS_NO_PREEMPT`), the task will retain control of the processor as long as it is in the executing state – even if a higher priority task is made ready. If preemption is enabled (`RTEMS_PREEMPT`) and a higher priority task is made ready, then the processor will be taken away from the current task immediately and given to the higher priority task.

The timeslicing component is used by the RTEMS scheduler to determine how the processor is allocated to tasks of equal priority. If timeslicing is enabled (`RTEMS_TIMESLICE`), then RTEMS will limit the amount of time the task can execute before the processor is allocated to another ready task of equal priority. The length of the timeslice is application dependent and specified in the Configuration Table. If timeslicing is disabled (`RTEMS_NO_TIMESLICE`), then the task will be allowed to execute until a task of higher priority is made ready. If `RTEMS_NO_PREEMPT` is selected, then the timeslicing component is ignored by the scheduler.

The asynchronous signal processing component is used to determine when received signals are to be processed by the task. If signal processing is enabled (`RTEMS_ASR`), then signals sent to the task will be processed the next time the task executes. If signal processing is disabled (`RTEMS_NO_ASR`), then all signals received by the task will remain posted until signal processing is enabled. This component affects only tasks which have established a routine to process asynchronous signals.

The interrupt level component is used to determine which interrupts will be enabled when the task is executing. `RTEMS_INTERRUPT_LEVEL(n)` specifies that the task will execute at interrupt level n.

- `RTEMS_PREEMPT` - enable preemption (default)
- `RTEMS_NO_PREEMPT` - disable preemption
- `RTEMS_NO_TIMESLICE` - disable timeslicing (default)
- `RTEMS_TIMESLICE` - enable timeslicing
- `RTEMS_ASR` - enable ASR processing (default)
- `RTEMS_NO_ASR` - disable ASR processing

- RTEMS_INTERRUPT_LEVEL(0) - enable all interrupts (default)
- RTEMS_INTERRUPT_LEVEL(n) - execute at interrupt level n

4.2.6 Accessing Task Arguments

All RTEMS tasks are invoked with a single argument which is specified when they are started or restarted. The argument is commonly used to communicate startup information to the task. The simplest manner in which to define a task which accesses it argument is:

```
rtems_task user_task(
  rtems_task_argument argument
);
```

Application tasks requiring more information may view this single argument as an index into an array of parameter blocks.

4.2.7 Floating Point Considerations

Creating a task with the RTEMS_FLOATING_POINT flag results in additional memory being allocated for the TCB to store the state of the numeric coprocessor during task switches. This additional memory is **NOT** allocated for RTEMS_NO_FLOATING_POINT tasks. Saving and restoring the context of a RTEMS_FLOATING_POINT task takes longer than that of a RTEMS_NO_FLOATING_POINT task because of the relatively large amount of time required for the numeric coprocessor to save or restore its computational state.

Since RTEMS was designed specifically for embedded military applications which are floating point intensive, the executive is optimized to avoid unnecessarily saving and restoring the state of the numeric coprocessor. The state of the numeric coprocessor is only saved when a RTEMS_FLOATING_POINT task is dispatched and that task was not the last task to utilize the coprocessor. In a system with only one RTEMS_FLOATING_POINT task, the state of the numeric coprocessor will never be saved or restored.

Although the overhead imposed by RTEMS_FLOATING_POINT tasks is minimal, some applications may wish to completely avoid the overhead associated with RTEMS_FLOATING_POINT tasks and still utilize a numeric coprocessor. By preventing a task from being preempted while performing a sequence of floating point operations, a RTEMS_NO_FLOATING_POINT task can utilize the numeric coprocessor without incurring the overhead of a RTEMS_FLOATING_POINT context switch. This approach also avoids the allocation of a floating point context area. However, if this approach is taken by the application designer, NO tasks should be created as RTEMS_FLOATING_POINT tasks. Otherwise, the floating point context will not be correctly maintained because RTEMS assumes that the state of the numeric coprocessor will not be altered by RTEMS_NO_FLOATING_POINT tasks.

If the supported processor type does not have hardware floating capabilities or a standard numeric coprocessor, RTEMS will not provide built-in support for hardware floating point on that processor. In this case, all tasks are considered RTEMS_NO_FLOATING_POINT whether created as

RTEMS_FLOATING_POINT or RTEMS_NO_FLOATING_POINT tasks. A floating point emulation software library must be utilized for floating point operations.

On some processors, it is possible to disable the floating point unit dynamically. If this capability is supported by the target processor, then RTEMS will utilize this capability to enable the floating point unit only for tasks which are created with the RTEMS_FLOATING_POINT attribute. The consequence of a RTEMS_NO_FLOATING_POINT task attempting to access the floating point unit is CPU dependent but will generally result in an exception condition.

4.2.8 Building a Task's Attribute Set

In general, an attribute set is built by a bitwise OR of the desired components. The set of valid task attribute components is listed below:

- RTEMS_NO_FLOATING_POINT - does not use coprocessor (default)
- RTEMS_FLOATING_POINT - uses numeric coprocessor
- RTEMS_LOCAL - local task (default)
- RTEMS_GLOBAL - global task

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. A component listed as a default is not required to appear in the component list, although it is a good programming practice to specify default components. If all defaults are desired, then RTEMS_DEFAULT_ATTRIBUTES should be used.

This example demonstrates the attribute_set parameter needed to create a local task which utilizes the numeric coprocessor. The attribute_set parameter could be RTEMS_FLOATING_POINT or RTEMS_LOCAL | RTEMS_FLOATING_POINT. The attribute_set parameter can be set to RTEMS_FLOATING_POINT because RTEMS_LOCAL is the default for all created tasks. If the task were global and used the numeric coprocessor, then the attribute_set parameter would be RTEMS_GLOBAL | RTEMS_FLOATING_POINT.

4.2.9 Building a Mode and Mask

In general, a mode and its corresponding mask is built by a bitwise OR of the desired components. The set of valid mode constants and each mode's corresponding mask constant is listed below:

- RTEMS_PREEMPT is masked by RTEMS_PREEMPT_MASK and enables preemption
- RTEMS_NO_PREEMPT is masked by RTEMS_PREEMPT_MASK and disables preemption
- RTEMS_NO_TIMESLICE is masked by RTEMS_TIMESLICE_MASK and disables timeslicing
- RTEMS_TIMESLICE is masked by RTEMS_TIMESLICE_MASK and enables timeslicing
- RTEMS_ASR is masked by RTEMS_ASR_MASK and enables ASR processing

- `RTEMS_NO_ASR` is masked by `RTEMS_ASR_MASK` and disables ASR processing
- `RTEMS_INTERRUPT_LEVEL(0)` is masked by `RTEMS_INTERRUPT_MASK` and enables all interrupts
- `RTEMS_INTERRUPT_LEVEL(n)` is masked by `RTEMS_INTERRUPT_MASK` and sets interrupts level n

Mode values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each mode appears exactly once in the component list. A mode component listed as a default is not required to appear in the mode component list, although it is a good programming practice to specify default components. If all defaults are desired, the mode `RTEMS_DEFAULT_MODES` and the mask `RTEMS_ALL_MODE_MASKS` should be used.

The following example demonstrates the mode and mask parameters used with the `rtems_task_mode` directive to place a task at interrupt level 3 and make it non-preemptible. The mode should be set to `RTEMS_INTERRUPT_LEVEL(3) | RTEMS_NO_PREEMPT` to indicate the desired preemption mode and interrupt level, while the mask parameter should be set to `RTEMS_INTERRUPT_MASK | RTEMS_NO_PREEMPT_MASK` to indicate that the calling task's interrupt level and preemption mode are being altered.

4.3 Operations

4.3.1 Creating Tasks

The `rtems_task_create` directive creates a task by allocating a task control block, assigning the task a user-specified name, allocating it a stack and floating point context area, setting a user-specified initial priority, setting a user-specified initial mode, and assigning it a task ID. Newly created tasks are initially placed in the dormant state. All RTEMS tasks execute in the most privileged mode of the processor.

4.3.2 Obtaining Task IDs

When a task is created, RTEMS generates a unique task ID and assigns it to the created task until it is deleted. The task ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_task_create` directive, the task ID is stored in a user provided location. Second, the task ID may be obtained later using the `rtems_task_ident` directive. The task ID is used by other directives to manipulate this task.

4.3.3 Starting and Restarting Tasks

The `rtems_task_start` directive is used to place a dormant task in the ready state. This enables the task to compete, based on its current priority, for the processor and other system resources.

Any actions, such as suspension or change of priority, performed on a task prior to starting it are nullified when the task is started.

With the `rtems_task_start` directive the user specifies the task's starting address and argument. The argument is used to communicate some startup information to the task. As part of this directive, RTEMS initializes the task's stack based upon the task's initial execution mode and start address. The starting argument is passed to the task in accordance with the target processor's calling convention.

The `rtems_task_restart` directive restarts a task at its initial starting address with its original priority and execution mode, but with a possibly different argument. The new argument may be used to distinguish between the original invocation of the task and subsequent invocations. The task's stack and control block are modified to reflect their original creation values. Although references to resources that have been requested are cleared, resources allocated by the task are NOT automatically returned to RTEMS. A task cannot be restarted unless it has previously been started (i.e. dormant tasks cannot be restarted). All restarted tasks are placed in the ready state.

4.3.4 Suspending and Resuming Tasks

The `rtems_task_suspend` directive is used to place either the caller or another task into a suspended state. The task remains suspended until a `rtems_task_resume` directive is issued. This implies that a task may be suspended as well as blocked waiting either to acquire a resource or for the expiration of a timer.

The `rtems_task_resume` directive is used to remove another task from the suspended state. If the task is not also blocked, resuming it will place it in the ready state, allowing it to once again compete for the processor and resources. If the task was blocked as well as suspended, this directive clears the suspension and leaves the task in the blocked state.

Suspending a task which is already suspended or resuming a task which is not suspended is considered an error.

4.3.5 Delaying the Currently Executing Task

The `rtems_task_wake_after` directive creates a sleep timer which allows a task to go to sleep for a specified interval. The task is blocked until the delay interval has elapsed, at which time the task is unblocked. A task calling the `rtems_task_wake_after` directive with a delay interval of `RTEMS_YIELD_PROCESSOR` ticks will yield the processor to any other ready task of equal or greater priority and remain ready to execute.

The `rtems_task_wake_when` directive creates a sleep timer which allows a task to go to sleep until a specified date and time. The calling task is blocked until the specified date and time has occurred, at which time the task is unblocked.

4.3.6 Changing Task Priority

The `rtems_task_set_priority` directive is used to obtain or change the current priority of either the calling task or another task. If the new priority requested is `RTEMS_CURRENT_PRIORITY` or the task's actual priority, then the current priority will be returned and the task's priority will remain unchanged. If the task's priority is altered, then the task will be scheduled according to its new priority.

The `rtems_task_restart` directive resets the priority of a task to its original value.

4.3.7 Changing Task Mode

The `rtems_task_mode` directive is used to obtain or change the current execution mode of the calling task. A task's execution mode is used to enable preemption, timeslicing, ASR processing, and to set the task's interrupt level.

The `rtems_task_restart` directive resets the mode of a task to its original value.

4.3.8 Notepad Locations

RTEMS provides sixteen notepad locations for each task. Each notepad location may contain a note consisting of four bytes of information. RTEMS provides two directives, `rtems_task_set_note` and `rtems_task_get_note`, that enable a user to access and change the notepad locations. The `rtems_task_set_note` directive enables the user to set a task's notepad entry to a specified note. The `rtems_task_get_note` directive allows the user to obtain the note contained in any one of the sixteen notepads of a specified task.

4.3.9 Task Deletion

RTEMS provides the `rtems_task_delete` directive to allow a task to delete itself or any other task. This directive removes all RTEMS references to the task, frees the task's control block, removes it from resource wait queues, and deallocates its stack as well as the optional floating point context. The task's name and ID become inactive at this time, and any subsequent references to either of them is invalid. In fact, RTEMS may reuse the task ID for another task which is created later in the application.

Unexpired delay timers (i.e. those used by `rtems_task_wake_after` and `rtems_task_wake_when`) and timeout timers associated with the task are automatically deleted, however, other resources dynamically allocated by the task are NOT automatically returned to RTEMS. Therefore, before a task is deleted, all of its dynamically allocated resources should be deallocated by the user. This may be accomplished by instructing the task to delete itself rather than directly deleting the task. Other tasks may instruct a task to delete itself by sending a "delete self" message, event, or signal, or by restarting the task with special arguments which instruct the task to delete itself.

4.4 Directives

This section details the task manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

4.4.1 TASK_CREATE - Create a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_create(
  rtems_name          name,
  rtems_task_priority  initial_priority,
  rtems_unsigned32     stack_size,
  rtems_mode           initial_modes,
  rtems_attribute      attribute_set,
  rtems_id            *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task created successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_INVALID_SIZE` - stack too small
`RTEMS_INVALID_PRIORITY` - invalid task priority
`RTEMS_MP_NOT_CONFIGURED` - multiprocessing not configured
`RTEMS_TOO_MANY` - too many tasks created
`RTEMS_UNSATISFIED` - not enough memory for stack/FP context
`RTEMS_TOO_MANY` - too many global objects

DESCRIPTION:

This directive creates a task which resides on the local node. It allocates and initializes a TCB, a stack, and an optional floating point context area. The mode parameter contains values which sets the task's initial execution mode. The `RTEMS_FLOATING_POINT` attribute should be specified if the created task is to use a numeric coprocessor. For performance reasons, it is recommended that tasks not using the numeric coprocessor should specify the `RTEMS_NO_FLOATING_POINT` attribute. If the `RTEMS_GLOBAL` attribute is specified, the task can be accessed from remote nodes. The task id, returned in id, is used in other task related directives to access the task. When created, a task is placed in the dormant state and can only be made ready to execute using the directive `rtems_task_start`.

NOTES:

This directive will not cause the calling task to be preempted.

Valid task priorities range from a high of 1 to a low of 255.

RTEMS supports a maximum of 256 interrupt levels which are mapped onto the interrupt levels actually supported by the target processor.

The requested stack size should be at least `RTEMS_MINIMUM_STACK_SIZE` bytes. The value of `RTEMS_MINIMUM_STACK_SIZE` is processor dependent. Application developers should consider the stack usage of the device drivers when calculating the stack size required for tasks which utilize the driver.

The following task attribute constants are defined by RTEMS:

- `RTEMS_NO_FLOATING_POINT` - does not use coprocessor (default)
- `RTEMS_FLOATING_POINT` - uses numeric coprocessor
- `RTEMS_LOCAL` - local task (default)
- `RTEMS_GLOBAL` - global task

The following task mode constants are defined by RTEMS:

- `RTEMS_PREEMPT` - enable preemption (default)
- `RTEMS_NO_PREEMPT` - disable preemption
- `RTEMS_NO_TIMESLICE` - disable timeslicing (default)
- `RTEMS_TIMESLICE` - enable timeslicing
- `RTEMS_ASR` - enable ASR processing (default)
- `RTEMS_NO_ASR` - disable ASR processing
- `RTEMS_INTERRUPT_LEVEL(0)` - enable all interrupts (default)
- `RTEMS_INTERRUPT_LEVEL(n)` - execute at interrupt level n

Tasks should not be made global unless remote tasks must interact with them. This avoids the system overhead incurred by the creation of a global task. When a global task is created, the task's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including tasks, is limited by the maximum_global_objects field in the Configuration Table.

4.4.2 TASK_IDENT - Get ID of a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_ident(
  rtems_name        name,
  rtems_unsigned32  node,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task identified successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_INVALID_NODE` - invalid node id

DESCRIPTION:

This directive obtains the task id associated with the task name specified in name. A task may obtain its own id by specifying `RTEMS_SELF` or its own task name in name. If the task name is not unique, then the task id returned will match one of the tasks with that name. However, this task id is not guaranteed to correspond to the desired task. The task id, returned in id, is used in other task related directives to access the task.

NOTES:

This directive will not cause the running task to be preempted.

If node is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If node is a valid node number which does not represent the local node, then only the tasks exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

4.4.3 TASK_START - Start a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_start(
  rtems_id            id,
  rtems_task_entry    entry_point,
  rtems_task_argument argument
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - ask started successfully
`RTEMS_INVALID_ADDRESS` - invalid task entry point
`RTEMS_INVALID_ID` - invalid task id
`RTEMS_INCORRECT_STATE` - task not in the dormant state
`RTEMS_ILLEGAL_ON_REMOTE_OBJECT` - cannot start remote task

DESCRIPTION:

This directive readies the task, specified by tid, for execution based on the priority and execution mode specified when the task was created. The starting address of the task is given in entry_point. The task's starting argument is contained in argument. This argument can be a single value or used as an index into an array of parameter blocks.

NOTES:

The calling task will be preempted if its preemption mode is enabled and the task being started has a higher priority.

Any actions performed on a dormant task such as suspension or change of priority are nullified when the task is initiated via the `rtems_task_start` directive.

4.4.4 TASK_RESTART - Restart a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_restart(
  rtems_id            id,
  rtems_task_argument argument
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task restarted successfully
`RTEMS_INVALID_ID` - task id invalid
`RTEMS_INCORRECT_STATE` - task never started
`RTEMS_ILLEGAL_ON_REMOTE_OBJECT` - cannot restart remote task

DESCRIPTION:

This directive resets the task specified by id to begin execution at its original starting address. The task's priority and execution mode are set to the original creation values. If the task is currently blocked, RTEMS automatically makes the task ready. A task can be restarted from any state, except the dormant state.

The task's starting argument is contained in argument. This argument can be a single value or an index into an array of parameter blocks. This new argument may be used to distinguish between the initial `rtems_task_start` of the task and any ensuing calls to `rtems_task_restart` of the task. This can be beneficial in deleting a task. Instead of deleting a task using the `rtems_task_delete` directive, a task can delete another task by restarting that task, and allowing that task to release resources back to RTEMS and then delete itself.

NOTES:

If id is `RTEMS_SELF`, the calling task will be restarted and will not return from this directive.

The calling task will be preempted if its preemption mode is enabled and the task being restarted has a higher priority.

The task must reside on the local node, even if the task was created with the `RTEMS_GLOBAL` option.

4.4.5 TASK_DELETE - Delete a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task restarted successfully
`RTEMS_INVALID_ID` - task id invalid
`RTEMS_ILLEGAL_ON_REMOTE_OBJECT` - cannot restart remote task

DESCRIPTION:

This directive deletes a task, either the calling task or another task, as specified by id. RTEMS stops the execution of the task and reclaims the stack memory, any allocated delay or timeout timers, the TCB, and, if the task is `RTEMS_FLOATING_POINT`, its floating point context area. RTEMS does not reclaim the following resources: region segments, partition buffers, semaphores, timers, or rate monotonic periods.

NOTES:

A task is responsible for releasing its resources back to RTEMS before deletion. To insure proper deallocation of resources, a task should not be deleted unless it is unable to execute or does not hold any RTEMS resources. If a task holds RTEMS resources, the task should be allowed to deallocate its resources before deletion. A task can be directed to release its resources and delete itself by restarting it with a special argument or by sending it a message, an event, or a signal.

Deletion of the current task (`RTEMS_SELF`) will force RTEMS to select another task to execute.

When a global task is deleted, the task id must be transmitted to every node in the system for deletion from the local copy of the global object table.

The task must reside on the local node, even if the task was created with the `RTEMS_GLOBAL` option.

4.4.6 TASK_SUSPEND - Suspend a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_suspend(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task restarted successfully
`RTEMS_INVALID_ID` - task id invalid
`RTEMS_ALREADY_SUSPENDED` - task already suspended

DESCRIPTION:

This directive suspends the task specified by id from further execution by placing it in the suspended state. This state is additive to any other blocked state that the task may already be in. The task will not execute again until another task issues the `rtems_task_resume` directive for this task and any blocked state has been removed.

NOTES:

The requesting task can suspend itself by specifying `RTEMS_SELF` as id. In this case, the task will be suspended and a successful return code will be returned when the task is resumed.

Suspending a global task which does not reside on the local node will generate a request to the remote node to suspend the specified task.

If the task specified by id is already suspended, then the `RTEMS_ALREADY_SUSPENDED` status code is returned.

4.4.7 TASK_RESUME - Resume a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_resume(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task restarted successfully
`RTEMS_INVALID_ID` - task id invalid
`RTEMS_INCORRECT_STATE` - task not suspended

DESCRIPTION:

This directive removes the task specified by id from the suspended state. If the task is in the ready state after the suspension is removed, then it will be scheduled to run. If the task is still in a blocked state after the suspension is removed, then it will remain in that blocked state.

NOTES:

The running task may be preempted if its preemption mode is enabled and the local task being resumed has a higher priority.

Resuming a global task which does not reside on the local node will generate a request to the remote node to resume the specified task.

If the task specified by id is not suspended, then the `RTEMS_INCORRECT_STATE` status code is returned.

4.4.8 TASK_SET_PRIORITY - Set task priority

CALLING SEQUENCE:

```
rtems_status_code rtems_task_set_priority(
  rtems_id             id,
  rtems_task_priority  new_priority,
  rtems_task_priority *old_priority
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task priority set successfully
`RTEMS_INVALID_ID` - invalid task id
`RTEMS_INVALID_PRIORITY` - invalid task priority

DESCRIPTION:

This directive manipulates the priority of the task specified by id. An id of `RTEMS_SELF` is used to indicate the calling task. When new_priority is not equal to `RTEMS_CURRENT_PRIORITY`, the specified task's previous priority is returned in old_priority. When new_priority is `RTEMS_CURRENT_PRIORITY`, the specified task's current priority is returned in old_priority. Valid priorities range from a high of 1 to a low of 255.

NOTES:

The calling task may be preempted if its preemption mode is enabled and it lowers its own priority or raises another task's priority.

Setting the priority of a global task which does not reside on the local node will generate a request to the remote node to change the priority of the specified task.

If the task specified by id is currently holding any binary semaphores which use the priority inheritance algorithm, then the task's priority cannot be lowered immediately. If the task's priority were lowered immediately, then priority inversion results. The requested lowering of the task's priority will occur when the task has released all priority inheritance binary semaphores. The task's priority can be increased regardless of the task's use of priority inheritance binary semaphores.

4.4.9 TASK_MODE - Change current task's mode

CALLING SEQUENCE:

```
rtems_status_code rtems_task_mode(
  rtems_mode  mode_set,
  rtems_mode  mask,
  rtems_mode *previous_mode_set
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task mode set successfully

DESCRIPTION:

This directive manipulates the execution mode of the calling task. A task's execution mode enables and disables preemption, timeslicing, asynchronous signal processing, as well as specifying the current interrupt level. To modify an execution mode, the mode class(es) to be changed must be specified in the mask parameter and the desired mode(s) must be specified in the mode parameter.

NOTES:

The calling task will be preempted if it enables preemption and a higher priority task is ready to run.

Enabling timeslicing has no effect if preemption is enabled.

A task can obtain its current execution mode, without modifying it, by calling this directive with a mask value of `RTEMS_CURRENT_MODE`.

To temporarily disable the processing of a valid ASR, a task should call this directive with the `RTEMS_NO_ASR` indicator specified in mode.

The set of task mode constants and each mode's corresponding mask constant is provided in the following table:

- `RTEMS_PREEMPT` is masked by `RTEMS_PREEMPT_MASK` and enables preemption
- `RTEMS_NO_PREEMPT` is masked by `RTEMS_PREEMPT_MASK` and disables preemption
- `RTEMS_NO_TIMESLICE` is masked by `RTEMS_TIMESLICE_MASK` and disables timeslicing
- `RTEMS_TIMESLICE` is masked by `RTEMS_TIMESLICE_MASK` and enables timeslicing
- `RTEMS_ASR` is masked by `RTEMS_ASR_MASK` and enables ASR processing
- `RTEMS_NO_ASR` is masked by `RTEMS_ASR_MASK` and disables ASR processing

- `RTEMS_INTERRUPT_LEVEL(0)` is masked by `RTEMS_INTERRUPT_MASK` and enables all interrupts
- `RTEMS_INTERRUPT_LEVEL(n)` is masked by `RTEMS_INTERRUPT_MASK` and sets interrupts level n

4.4.10 TASK_GET_NOTE - Get task notepad entry

CALLING SEQUENCE:

```
rtems_status_code rtems_task_get_note(
  rtems_id          id,
  rtems_unsigned32  notepad,
  rtems_unsigned32 *note
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - note obtained successfully
`RTEMS_INVALID_ID` - invalid task id
`RTEMS_INVALID_NUMBER` - invalid notepad location

DESCRIPTION:

This directive returns the note contained in the notepad location of the task specified by id.

NOTES:

This directive will not cause the running task to be preempted.

If id is set to `RTEMS_SELF`, the calling task accesses its own notepad.

The sixteen notepad locations can be accessed using the constants `RTEMS_NOTEPAD_0` through `RTEMS_NOTEPAD_15`.

Getting a note of a global task which does not reside on the local node will generate a request to the remote node to obtain the notepad entry of the specified task.

4.4.11 TASK_SET_NOTE - Set task notepad entry

CALLING SEQUENCE:

```
rtems_status_code rtems_task_set_note(
  rtems_id         id,
  rtems_unsigned32 notepad,
  rtems_unsigned32 note
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - task's note set successfully
`RTEMS_INVALID_ID` - invalid task id
`RTEMS_INVALID_NUMBER` - invalid notepad location

DESCRIPTION:

This directive sets the notepad entry for the task specified by id to the value note.

NOTES:

If id is set to `RTEMS_SELF`, the calling task accesses its own notepad locations.

This directive will not cause the running task to be preempted.

The sixteen notepad locations can be accessed using the constants `RTEMS_NOTEPAD_0` through `RTEMS_NOTEPAD_15`.

Setting a notepad location of a global task which does not reside on the local node will generate a request to the remote node to set the specified notepad entry.

4.4.12 TASK_WAKE_AFTER - Wake up after interval

CALLING SEQUENCE:

```
rtems_status_code rtems_task_wake_after(
  rtems_interval ticks
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - always successful

DESCRIPTION:

This directive blocks the calling task for the specified number of system clock ticks. When the requested interval has elapsed, the task is made ready. The `rtems_clock_tick` directive automatically updates the delay period.

NOTES:

Setting the system date and time with the `rtems_clock_set` directive has no effect on a `rtems_task_wake_after` blocked task.

A task may give up the processor and remain in the ready state by specifying a value of `RTEMS_YIELD_PROCESSOR` in ticks.

The maximum timer interval that can be specified is the maximum value which can be represented by the rtems_unsigned32 type.

A clock tick is required to support the functionality of this directive.

4.4.13 TASK_WAKE_WHEN - Wake up when specified

CALLING SEQUENCE:

```
rtems_status_code rtems_task_wake_when(
  rtems_time_of_day *time_buffer
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - awakened at date/time successfully
`INVALID_TIME_OF_DAY` - invalid time buffer
`RTEMS_NOT_DEFINED` - system date and time is not set

DESCRIPTION:

This directive blocks a task until the date and time specified in time_buffer. At the requested date and time, the calling task will be unblocked and made ready to execute.

NOTES:

The ticks portion of time_buffer structure is ignored. The timing granularity of this directive is a second.

A clock tick is required to support the functionality of this directive.

5 Interrupt Manager

5.1 Introduction

Any real-time executive must provide a mechanism for quick response to externally generated interrupts to satisfy the critical time constraints of the application. The interrupt manager provides this mechanism for RTEMS. This manager permits quick interrupt response times by providing the critical ability to alter task execution which allows a task to be preempted upon exit from an ISR. The interrupt manager includes the following directive:

- `rtems_interrupt_catch` - Establish an ISR
- `rtems_interrupt_disable` - Disable Interrupts
- `rtems_interrupt_enable` - Enable Interrupts
- `rtems_interrupt_flash` - Flash Interrupt
- `rtems_interrupt_is_in_progress` - Is an ISR in Progress

5.2 Background

5.2.1 Processing an Interrupt

The interrupt manager allows the application to connect a function to a hardware interrupt vector. When an interrupt occurs, the processor will automatically vector to RTEMS. RTEMS saves and restores all registers which are not preserved by the normal C calling convention for the target processor and invokes the user's ISR. The user's ISR is responsible for processing the interrupt, clearing the interrupt if necessary, and device specific manipulation.

The `rtems_interrupt_catch` directive connects a procedure to an interrupt vector. The interrupt service routine is assumed to abide by these conventions and have a prototype similar to the following:

```
rtems_isr user_isr(
  rtems_vector_number vector
);
```

The vector number argument is provided by RTEMS to allow the application to identify the interrupt source. This could be used to allow a single routine to service interrupts from multiple instances of the same device. For example, a single routine could service interrupts from multiple serial ports and use the vector number to identify which port requires servicing.

To minimize the masking of lower or equal priority level interrupts, the ISR should perform the minimum actions required to service the interrupt. Other non-essential actions should be handled by application tasks. Once the user's ISR has completed, it returns control to the RTEMS interrupt manager which will perform task dispatching and restore the registers saved before the ISR was invoked.

The RTEMS interrupt manager guarantees that proper task scheduling and dispatching are performed at the conclusion of an ISR. A system call made by the ISR may have readied a task of higher priority than the interrupted task. Therefore, when the ISR completes, the postponed dispatch processing must be performed. No dispatch processing is performed as part of directives which have been invoked by an ISR.

Applications must adhere to the following rule if proper task scheduling and dispatching is to be performed:

> **The interrupt manager must be used for all ISRs which may be interrupted by the highest priority ISR which invokes an RTEMS directive.**

Consider a processor which allows a numerically low interrupt level to interrupt a numerically greater interrupt level. In this example, if an RTEMS directive is used in a level 4 ISR, then all ISRs which execute at levels 0 through 4 must use the interrupt manager.

Interrupts are nested whenever an interrupt occurs during the execution of another ISR. RTEMS supports efficient interrupt nesting by allowing the nested ISRs to terminate without performing any dispatch processing. Only when the outermost ISR terminates will the postponed dispatching occur.

5.2.2 RTEMS Interrupt Levels

Many processors support multiple interrupt levels or priorities. The exact number of interrupt levels is processor dependent. RTEMS internally supports 256 interrupt levels which are mapped to the processor's interrupt levels. For specific information on the mapping between RTEMS and the target processor's interrupt levels, refer to the Interrupt Processing chapter of the Applications Supplement document for a specific target processor.

5.2.3 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables all maskable interrupts before the execution of the section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for a minimum length of time. The maximum length of time interrupts are disabled by RTEMS is processor dependent and is detailed in the Timing Specification chapter of the Applications Supplement document for a specific target processor.

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

5.3 Operations

5.3.1 Establishing an ISR

The `rtems_interrupt_catch` directive establishes an ISR for the system. The address of the ISR and its associated CPU vector number are specified to this directive. This directive installs the RTEMS interrupt wrapper in the processor's Interrupt Vector Table and the address of the user's ISR in the RTEMS' Vector Table. This directive returns the previous contents of the specified vector in the RTEMS' Vector Table.

5.3.2 Directives Allowed from an ISR

Using the interrupt manager insures that RTEMS knows when a directive is being called from an ISR. The ISR may then use system calls to synchronize itself with an application task. The synchronization may involve messages, events or signals being passed by the ISR to the desired task. Directives invoked by an ISR must operate only on objects which reside on the local node. The following is a list of RTEMS system calls that may be made from an ISR:

- Task Management
    - task_get_note, task_set_note, task_suspend, task_resume
- Clock Management
    - clock_get, clock_tick
- Message, Event, and Signal Management
    - message_queue_send, message_queue_urgent
    - event_send
    - signal_send
- Semaphore Management
    - semaphore_release
- Dual-Ported Memory Management
    - port_external_to_internal, port_internal_to_external
- IO Management
    - io_initialize, io_open, io_close, io_read, io_write, io_control
- Fatal Error Management
    - fatal_error_occurred
- Multiprocessing
    - multiprocessing_announce

5.4 Directives

This section details the interrupt manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

5.4.1 INTERRUPT_CATCH - Establish an ISR

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_catch(
  rtems_isr_entry      new_isr_handler,
  rtems_vector_number  vector,
  rtems_isr_entry     *old_isr_handler
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - ISR established successfully
`RTEMS_INVALID_NUMBER` - illegal vector number
`RTEMS_INVALID_ADDRESS` - illegal ISR entry point or invalid old_isr_handler

DESCRIPTION:

This directive establishes an interrupt service routine (ISR) for the specified interrupt vector number. The `new_isr_handler` parameter specifies the entry point of the ISR. The entry point of the previous ISR for the specified vector is returned in `old_isr_handler`.

NOTES:

This directive will not cause the calling task to be preempted.

5.4.2 INTERRUPT_DISABLE - Disable Interrupts

CALLING SEQUENCE:

```
void rtems_interrupt_disable(
  rtems_isr_level  level
);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive disables all maskable interrupts and returns the previous `level`. A later invocation of the `rtems_interrupt_enable` directive should be used to restore the interrupt level.

NOTES:

This directive will not cause the calling task to be preempted.

This directive is implemented as a macro which modifies the `level` parameter.

5.4.3 INTERRUPT_ENABLE - Enable Interrupts

CALLING SEQUENCE:

```
void rtems_interrupt_enable(
  rtems_isr_level  level
);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive enables maskable interrupts to the `level` which was returned by a previous call to `rtems_interrupt_disable`. Immediately prior to invoking this directive, maskable interrupts should be disabled by a call to `rtems_interrupt_disable` and will be enabled when this directive returns to the caller.

NOTES:

This directive will not cause the calling task to be preempted.

5.4.4 INTERRUPT_FLASH - Flash Interrupts

CALLING SEQUENCE:

```
void rtems_interrupt_flash(
  rtems_isr_level level
);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive temporarily enables maskable interrupts to the `level` which was returned by a previous call to `rtems_interrupt_disable`. Immediately prior to invoking this directive, maskable interrupts should be disabled by a call to `rtems_interrupt_disable` and will be redisabled when this directive returns to the caller.

NOTES:

This directive will not cause the calling task to be preempted.

5.4.5 INTERRUPT_IS_IN_PROGRESS - Is an ISR in Progress

CALLING SEQUENCE:

```
    rtems_boolean rtems_interrupt_is_in_progress( void );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive returns `TRUE` if the processor is currently servicing an interrupt and `FALSE` otherwise. A return value of `TRUE` indicates that the caller is an interrupt service routine, **NOT** a task. The directives available to an interrupt service routine are restricted.

NOTES:

This directive will not cause the calling task to be preempted.

6 Clock Manager

6.1 Introduction

The clock manager provides support for time of day and other time related capabilities. The directives provided by the clock manager are:

- `rtems_clock_set` - Set system date and time
- `rtems_clock_get` - Get system date and time information
- `rtems_clock_tick` - Announce a clock tick

6.2 Background

6.2.1 Required Support

For the features provided by the clock manager to be utilized, periodic timer interrupts are required. Therefore, a real-time clock or hardware timer is necessary to create the timer interrupts. The `rtems_clock_tick` directive is normally called by the timer ISR to announce to RTEMS that a system clock tick has occurred. Elapsed time is measured in ticks. A tick is defined to be an integral number of microseconds which is specified by the user in the Configuration Table.

6.2.2 Time and Date Data Structures

The clock facilities of the clock manager operate upon calendar time. These directives utilize the following date and time structure for the native time and date format:

```
struct rtems_tod_control {
  rtems_unsigned32 year;   /* greater than 1987 */
  rtems_unsigned32 month;  /* 1 - 12 */
  rtems_unsigned32 day;    /* 1 - 31 */
  rtems_unsigned32 hour;   /* 0 - 23 */
  rtems_unsigned32 minute; /* 0 - 59 */
  rtems_unsigned32 second; /* 0 - 59 */
  rtems_unsigned32 ticks;  /* elapsed between seconds */
};

typedef struct rtems_tod_control rtems_time_of_day;
```

The native date and time format is the only format supported when setting the system date and time using the `rtems_clock_get` directive. Some applications expect to operate on a "UNIX-style" date and time data structure. The `rtems_clock_get` directive can optionally return the current date and time in the following structure:

```
typedef struct {
  rtems_unsigned32 seconds;       /* seconds since RTEMS epoch*/
  rtems_unsigned32 microseconds;  /* since last second       */
} rtems_clock_time_value;
```

The seconds field in this structure is the number of seconds since the RTEMS epoch of January 1, 1988.

6.2.3 Clock Tick and Timeslicing

Timeslicing is a task scheduling discipline in which tasks of equal priority are executed for a specific period of time before control of the CPU is passed to another task. It is also sometimes referred to as the automatic round-robin scheduling algorithm. The length of time allocated to each task is known as the quantum or timeslice.

The system's timeslice is defined as an integral number of ticks, and is specified in the Configuration Table. The timeslice is defined for the entire system of tasks, but timeslicing is enabled and disabled on a per task basis.

The `rtems_clock_tick` directive implements timeslicing by decrementing the running task's time-remaining counter when both timeslicing and preemption are enabled. If the task's timeslice has expired, then that task will be preempted if there exists a ready task of equal priority.

6.2.4 Delays

A sleep timer allows a task to delay for a given interval or up until a given time, and then wake and continue execution. This type of timer is created automatically by the `rtems_task_wake_after` and `rtems_task_wake_when` directives and, as a result, does not have an RTEMS ID. Once activated, a sleep timer cannot be explicitly deleted. Each task may activate one and only one sleep timer at a time.

6.2.5 Timeouts

Timeouts are a special type of timer automatically created when the timeout option is used on the `rtems_message_queue_receive`, `rtems_event_receive`, `rtems_semaphore_obtain` and `rtems_region_get_segment` directives. Each task may have one and only one timeout active at a time. When a timeout expires, it unblocks the task with a timeout status code.

6.3 Operations

6.3.1 Announcing a Tick

RTEMS provides the `rtems_clock_tick` directive which is called from the user's real-time clock ISR to inform RTEMS that a tick has elapsed. The tick frequency value, defined in microseconds, is a configuration parameter found in the Configuration Table. RTEMS divides one million microseconds (one second) by the number of microseconds per tick to determine the number of calls to the `rtems_clock_tick` directive per second. The frequency of `rtems_clock_tick` calls determines the resolution (granularity) for all time dependent RTEMS actions. For example, calling `rtems_clock_tick` ten times per second yields a higher resolution than calling `rtems_clock_tick` two times per second. The `rtems_clock_tick` directive is responsible for maintaining both calendar time and the dynamic set of timers.

6.3.2 Setting the Time

The `rtems_clock_set` directive allows a task or an ISR to set the date and time maintained by RTEMS. If setting the date and time causes any outstanding timers to pass their deadline, then the expired timers will be fired during the invocation of the `rtems_clock_set` directive.

6.3.3 Obtaining the Time

The `rtems_clock_get` directive allows a task or an ISR to obtain the current date and time or date and time related information. The current date and time can be returned in either native or UNIX-style format. Additionally, the application can obtain date and time related information such as the number of seconds since the RTEMS epoch, the number of ticks since the executive was initialized, and the number of ticks per second. The information returned by the `rtems_clock_get` directive is dependent on the option selected by the caller. The following options are available:

- `RTEMS_CLOCK_GET_TOD` - obtain native style date and time
- `RTEMS_CLOCK_GET_TIME_VALUE` - obtain UNIX-style date and time
- `RTEMS_CLOCK_GET_TICKS_SINCE_BOOT` - obtain number of ticks since RTEMS was initialized
- `RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH` - obtain number of seconds since RTEMS epoch
- `RTEMS_CLOCK_GET_TICKS_PER_SECOND` - obtain number of clock ticks per second

Calendar time operations will return an error code if invoked before the date and time have been set.

6.4 Directives

This section details the clock manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

6.4.1 CLOCK_SET - Set system date and time

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_set(
  rtems_time_of_day *time_buffer
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - date and time set successfully
`INVALID_TIME_OF_DAY` - invalid time of day

DESCRIPTION:

This directive sets the system date and time. The date, time, and ticks in the time_buffer structure are all range-checked, and an error is returned if any one is out of its valid range.

NOTES:

Years before 1988 are invalid.

The system date and time are based on the configured tick rate (number of microseconds in a tick).

Setting the time forward may cause a higher priority task, blocked waiting on a specific time, to be made ready. In this case, the calling task will be preempted after the next clock tick.

Re-initializing RTEMS causes the system date and time to be reset to an uninitialized state. Another call to `rtems_clock_set` is required to re-initialize the system date and time to application specific specifications.

6.4.2 CLOCK_GET - Get system date and time information

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_get(
  rtems_clock_get_options  option,
  void                    *time_buffer
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - current time obtained successfully
`RTEMS_NOT_DEFINED` - system date and time is not set

DESCRIPTION:

This directive obtains the system date and time. If the caller is attempting to obtain the date and time (i.e. option is set to either `RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH`, `RTEMS_CLOCK_GET_TOD`, or `RTEMS_CLOCK_GET_TIME_VALUE`) and the date and time has not been set with a previous call to `rtems_clock_set`, then the `RTEMS_NOT_DEFINED` status code is returned. The caller can always obtain the number of ticks per second (option is `RTEMS_CLOCK_GET_TICKS_PER_SECOND`) and the number of ticks since the executive was initialized option is `RTEMS_CLOCK_GET_TICKS_SINCE_BOOT`).

The data type expected for time_buffer is indicated below:

- `RTEMS_CLOCK_GET_TOD` - (rtems_time_of_day *)
- `RTEMS_CLOCK_GET_TIME_VALUE` - (rtems_clock_time_value *)
- `RTEMS_CLOCK_GET_TICKS_SINCE_BOOT` - (rtems_interval *)
- `RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH` - (rtems_interval *)
- `RTEMS_CLOCK_GET_TICKS_PER_SECOND` - (rtems_interval *)

NOTES:

This directive is callable from an ISR.

This directive will not cause the running task to be preempted. Re-initializing RTEMS causes the system date and time to be reset to an uninitialized state. Another call to `rtems_clock_set` is required to re-initialize the system date and time to application specific specifications.

6.4.3 CLOCK_TICK - Announce a clock tick

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_tick( void );
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - current time obtained successfully

DESCRIPTION:

This directive announces to RTEMS that a system clock tick has occurred. The directive is usually called from the timer interrupt ISR of the local processor. This directive maintains the system date and time, decrements timers for delayed tasks, timeouts, rate monotonic periods, and implements timeslicing.

NOTES:

This directive is typically called from an ISR.

The microseconds_per_tick and ticks_per_timeslice parameters in the Configuration Table contain the number of microseconds per tick and number of ticks per timeslice, respectively.

7 Timer Manager

7.1 Introduction

The timer manager provides support for timer facilities. The directives provided by the timer manager are:

- `rtems_timer_create` - Create a timer
- `rtems_timer_ident` - Get ID of a timer
- `rtems_timer_cancel` - Cancel a timer
- `rtems_timer_delete` - Delete a timer
- `rtems_timer_fire_after` - Fire timer after interval
- `rtems_timer_fire_when` - Fire timer when specified
- `rtems_timer_reset` - Reset an interval timer

7.2 Background

7.2.1 Required Support

A clock tick is required to support the functionality provided by this manager.

7.2.2 Timers

A timer is an RTEMS object which allows the application to schedule operations to occur at specific times in the future. User supplied timer service routines are invoked by the `rtems_clock_tick` directive when the timer fires. Timer service routines may perform any operations or directives which normally would be performed by the application code which invoked the `rtems_clock_tick` directive.

The timer can be used to implement watchdog routines which only fire to denote that an application error has occurred. The timer is reset at specific points in the application to insure that the watchdog does not fire. Thus, if the application does not reset the watchdog timer, then the timer service routine will fire to indicate that the application has failed to reach a reset point. This use of a timer is sometimes referred to as a "keep alive" or a "deadman" timer.

7.2.3 Timer Service Routines

The timer service routine should adhere to C calling conventions and have a prototype similar to the following:

```
rtems_timer_service_routine user_routine(
  rtems_id   timer_id,
  void      *user_data
);
```

Where the timer_id parameter is the RTEMS object ID of the timer which is being fired and user_data is a pointer to user-defined information which may be utilized by the timer service routine. The argument user_data may be NULL.

7.3 Operations

7.3.1 Creating a Timer

The `rtems_timer_create` directive creates a timer by allocating a Timer Control Block (TMCB), assigning the timer a user-specified name, and assigning it a timer ID. Newly created timers do not have a timer service routine associated with them and are not active.

7.3.2 Obtaining Timer IDs

When a timer is created, RTEMS generates a unique timer ID and assigns it to the created timer until it is deleted. The timer ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_timer_create` directive, the timer ID is stored in a user provided location. Second, the timer ID may be obtained later using the `rtems_timer_ident` directive. The timer ID is used by other directives to manipulate this timer.

7.3.3 Initiating an Interval Timer

The `rtems_timer_fire_after` directive initiates a timer to fire a user provided timer service routine after the specified number of clock ticks have elapsed. When the interval has elapsed, the timer service routine will be invoked from the `rtems_clock_tick` directive.

7.3.4 Initiating a Time of Day Timer

The `rtems_timer_fire_when` directive initiates a timer to fire a user provided timer service routine when the specified time of day has been reached. When the interval has elapsed, the timer service routine will be invoked from the `rtems_clock_tick` directive.

7.3.5 Canceling a Timer

The `rtems_timer_cancel` directive is used to halt the specified timer. Once canceled, the timer service routine will not fire unless the timer is reinitiated. The timer can be reinitiated using the `rtems_timer_reset`, `rtems_timer_fire_after`, and `rtems_timer_fire_when` directives.

7.3.6 Resetting a Timer

The `rtems_timer_reset` directive is used to restore an interval timer initiated by a previous invocation of `rtems_timer_fire_after` to its original interval length. If the timer has not been used or the last usage of this timer was by a `rtems_timer_fire_when` directive, then an error is returned. The timer service routine is not changed or fired by this directive.

7.3.7 Deleting a Timer

The `rtems_timer_delete` directive is used to delete a timer. If the timer is running and has not expired, the timer is automatically canceled. The timer's control block is returned to the TMCB free list when it is deleted. A timer can be deleted by a task other than the task which created the timer. Any subsequent references to the timer's name and ID are invalid.

7.4 Directives

This section details the timer manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

7.4.1 TIMER_CREATE - Create a timer

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_create(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - timer created successfully
`RTEMS_INVALID_NAME` - invalid timer name
`RTEMS_TOO_MANY` - too many timers created

DESCRIPTION:

This directive creates a timer. The assigned timer id is returned in id. This id is used to access the timer with other timer manager directives. For control and maintenance of the timer, RTEMS allocates a TMCB from the local TMCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

7.4.2 TIMER_IDENT - Get ID of a timer

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_ident(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - timer identified successfully
`RTEMS_INVALID_NAME` - timer name not found

DESCRIPTION:

This directive obtains the timer id associated with the timer name to be acquired. If the timer name is not unique, then the timer id will match one of the timers with that name. However, this timer id is not guaranteed to correspond to the desired timer. The timer id is used to access this timer in other timer related directives.

NOTES:

This directive will not cause the running task to be preempted.

7.4.3 TIMER_CANCEL - Cancel a timer

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_cancel(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - timer canceled successfully
`RTEMS_INVALID_ID` - invalid timer id

DESCRIPTION:

This directive cancels the timer id. This timer will be reinitiated by the next invocation of `rtems_timer_reset`, `rtems_timer_fire_after`, or `rtems_timer_fire_when` with id.

NOTES:

This directive will not cause the running task to be preempted.

7.4.4 TIMER_DELETE - Delete a timer

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - timer deleted successfully
`RTEMS_INVALID_ID` - invalid timer id

DESCRIPTION:

This directive deletes the timer specified by id. If the timer is running, it is automatically canceled. The TMCB for the deleted timer is reclaimed by RTEMS.

NOTES:

This directive will not cause the running task to be preempted.

A timer can be deleted by a task other than the task which created the timer.

7.4.5 TIMER_FIRE_AFTER - Fire timer after interval

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_fire_after(
  rtems_id                           id,
  rtems_interval                     ticks,
  rtems_timer_service_routine_entry  routine,
  void                              *user_data
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - timer initiated successfully
`RTEMS_INVALID_ID` - invalid timer id
`RTEMS_INVALID_NUMBER` - invalid interval

DESCRIPTION:

This directive initiates the timer specified by id. If the timer is running, it is automatically canceled before being initiated. The timer is scheduled to fire after an interval ticks clock ticks has passed. When the timer fires, the timer service routine routine will be invoked with the argument user_data.

NOTES:

This directive will not cause the running task to be preempted.

7.4.6 TIMER_FIRE_WHEN - Fire timer when specified

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_fire_when(
  rtems_id                           id,
  rtems_time_of_day                 *wall_time,
  rtems_timer_service_routine_entry  routine,
  void                              *user_data
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - timer initiated successfully
`RTEMS_INVALID_ID` - invalid timer id
`RTEMS_NOT_DEFINED` - system date and time is not set
`RTEMS_INVALID_CLOCK` - invalid time of day

DESCRIPTION:

This directive initiates the timer specified by id. If the timer is running, it is automatically canceled before being initiated. The timer is scheduled to fire at the time of day specified by wall_time. When the timer fires, the timer service routine routine will be invoked with the argument user_data.

NOTES:

This directive will not cause the running task to be preempted.

7.4.7 TIMER_RESET - Reset an interval timer

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_reset(
  rtems_id   id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - timer reset successfully
`RTEMS_INVALID_ID` - invalid timer id
`RTEMS_NOT_DEFINED` - attempted to reset a when or newly created timer

DESCRIPTION:

This directive resets the timer associated with id. This timer must have been previously initiated with a `rtems_timer_fire_after` directive. If active the timer is canceled, after which the timer is reinitiated using the same interval and timer service routine which the original `rtems_timer_fire_after` directive used.

NOTES:

If the timer has not been used or the last usage of this timer was by a `rtems_timer_fire_when` directive, then the `RTEMS_NOT_DEFINED` error is returned.

Restarting a cancelled after timer results in the timer being reinitiated with its previous timer service routine and interval.

This directive will not cause the running task to be preempted.

8 Semaphore Manager

8.1 Introduction

The semaphore manager utilizes standard Dijkstra counting semaphores to provide synchronization and mutual exclusion capabilities. The directives provided by the semaphore manager are:

- `rtems_semaphore_create` - Create a semaphore
- `rtems_semaphore_ident` - Get ID of a semaphore
- `rtems_semaphore_delete` - Delete a semaphore
- `rtems_semaphore_obtain` - Acquire a semaphore
- `rtems_semaphore_release` - Release a semaphore

8.2 Background

A semaphore can be viewed as a protected variable whose value can be modified only with the `rtems_semaphore_create`, `rtems_semaphore_obtain`, and `rtems_semaphore_release` directives. RTEMS supports both binary and counting semaphores. A binary semaphore is restricted to values of zero or one, while a counting semaphore can assume any non-negative integer value.

A binary semaphore can be used to control access to a single resource. In particular, it can be used to enforce mutual exclusion for a critical section in user code. In this instance, the semaphore would be created with an initial count of one to indicate that no task is executing the critical section of code. Upon entry to the critical section, a task must issue the `rtems_semaphore_obtain` directive to prevent other tasks from entering the critical section. Upon exit from the critical section, the task must issue the `rtems_semaphore_release` directive to allow another task to execute the critical section.

A counting semaphore can be used to control access to a pool of two or more resources. For example, access to three printers could be administered by a semaphore created with an initial count of three. When a task requires access to one of the printers, it issues the `rtems_semaphore_obtain` directive to obtain access to a printer. If a printer is not currently available, the task can wait for a printer to become available or return immediately. When the task has completed printing, it should issue the `rtems_semaphore_release` directive to allow other tasks access to the printer.

Task synchronization may be achieved by creating a semaphore with an initial count of zero. One task waits for the arrival of another task by issuing a `rtems_semaphore_obtain` directive when it reaches a synchronization point. The other task performs a corresponding `rtems_semaphore_release` operation when it reaches its synchronization point, thus unblocking the pending task.

8.2.1 Nested Resource Access

Deadlock occurs when a task owning a binary semaphore attempts to acquire that same semaphore and blocks as result. Since the semaphore is allocated to a task, it cannot be deleted. Therefore, the task that currently holds the semaphore and is also blocked waiting for that semaphore will never execute again.

RTEMS addresses this problem by allowing the task holding the binary semaphore to obtain the same binary semaphore multiple times in a nested manner. Each `rtems_semaphore_obtain` must be accompanied with a `rtems_semaphore_release`. The semaphore will only be made available for acquisition by other tasks when the outermost `rtems_semaphore_obtain` is matched with a `rtems_semaphore_release`.

8.2.2 Priority Inversion

Priority inversion is a form of indefinite postponement which is common in multitasking, preemptive executives with shared resources. Priority inversion occurs when a high priority tasks requests access to shared resource which is currently allocated to low priority task. The high priority task must block until the low priority task releases the resource. This problem is exacerbated when the low priority task is prevented from executing by one or more medium priority tasks. Because the low priority task is not executing, it cannot complete its interaction with the resource and release that resource. The high priority task is effectively prevented from executing by lower priority tasks.

8.2.3 Priority Inheritance

Priority inheritance is an algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task blocked waiting for that resource. Each time a task blocks attempting to obtain the resource, the task holding the resource may have its priority increased.

RTEMS supports priority inheritance for local, binary semaphores that use the priority task wait queue blocking discipline. When a task of higher priority than the task holding the semaphore blocks, the priority of the task holding the semaphore is increased to that of the blocking task. When the task holding the task completely releases the binary semaphore (i.e. not for a nested release), the holder's priority is restored to the value it had before any higher priority was inherited.

The RTEMS implementation of the priority inheritance algorithm takes into account the scenario in which a task holds more than one binary semaphore. The holding task will execute at the priority of the higher of the highest ceiling priority or at the priority of the highest priority task blocked waiting for any of the semaphores the task holds. Only when the task releases ALL of the binary semaphores it holds will its priority be restored to the normal value.

8.2.4 Priority Ceiling

Priority ceiling is an algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task which will EVER block waiting for that resource. This algorithm addresses the problem of priority inversion although it avoids the possibility of changing the priority of the task holding the resource multiple times. The priority ceiling algorithm will only change the priority of the task holding the resource a maximum of one time. The ceiling priority is set at creation time and must be the priority of the highest priority task which will ever attempt to acquire that semaphore.

RTEMS supports priority ceiling for local, binary semaphores that use the priority task wait queue blocking discipline. When a task of lower priority than the ceiling priority successfully obtains the semaphore, its priority is raised to the ceiling priority. When the task holding the task completely releases the binary semaphore (i.e. not for a nested release), the holder's priority is restored to the value it had before any higher priority was put into effect.

The need to identify the highest priority task which will attempt to obtain a particular semaphore can be a difficult task in a large, complicated system. Although the priority ceiling algorithm is more efficient than the priority inheritance algorithm with respect to the maximum number of task priority changes which may occur while a task holds a particular semaphore, the priority inheritance algorithm is more forgiving in that it does not require this apriori information.

The RTEMS implementation of the priority ceiling algorithm takes into account the scenario in which a task holds more than one binary semaphore. The holding task will execute at the priority of the higher of the highest ceiling priority or at the priority of the highest priority task blocked waiting for any of the semaphores the task holds. Only when the task releases ALL of the binary semaphores it holds will its priority be restored to the normal value.

8.2.5 Building a Semaphore's Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The following table lists the set of valid semaphore attributes:

- `RTEMS_FIFO` - tasks wait by FIFO (default)
- `RTEMS_PRIORITY` - tasks wait by priority
- `RTEMS_BINARY_SEMAPHORE` - restrict values to 0 and 1 (default)
- `RTEMS_COUNTING_SEMAPHORE` - no restriction on values
- `RTEMS_NO_INHERIT_PRIORITY` - do not use priority inheritance (default)
- `RTEMS_INHERIT_PRIORITY` - use priority inheritance
- `RTEMS_PRIORITY_CEILING` - use priority ceiling
- `RTEMS_NO_PRIORITY_CEILING` - do not use priority ceiling (default)
- `RTEMS_LOCAL` - local task (default)

- `RTEMS_GLOBAL` - global task

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute `RTEMS_DEFAULT_ATTRIBUTES` should be specified on this call.

This example demonstrates the attribute_set parameter needed to create a local semaphore with the task priority waiting queue discipline. The attribute_set parameter passed to the `rtems_semaphore_create` directive could be either `RTEMS_PRIORITY` or `RTEMS_LOCAL | RTEMS_PRIORITY`. The attribute_set parameter can be set to `RTEMS_PRIORITY` because `RTEMS_LOCAL` is the default for all created tasks. If a similar semaphore were to be known globally, then the attribute_set parameter would be `RTEMS_GLOBAL | RTEMS_PRIORITY`.

8.2.6 Building a SEMAPHORE_OBTAIN Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_semaphore_obtain` directive are listed in the following table:

- `RTEMS_WAIT` - task will wait for semaphore (default)
- `RTEMS_NO_WAIT` - task should not wait

Option values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An option listed as a default is not required to appear in the list, although it is a good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for a semaphore. The option parameter passed to the `rtems_semaphore_obtain` directive should be `RTEMS_NO_WAIT`.

8.3 Operations

8.3.1 Creating a Semaphore

The `rtems_semaphore_create` directive creates a binary or counting semaphore with a user-specified name as well as an initial count. If a binary semaphore is created with a count of zero (0) to indicate that it has been allocated, then the task creating the semaphore is considered the current holder of the semaphore. At create time the method for ordering waiting tasks in the semaphore's task wait queue (by FIFO or task priority) is specified. Additionally, the priority inheritance or priority ceiling algorithm may be selected for local, binary semaphores that use the priority task wait queue blocking discipline. If the priority ceiling algorithm is selected, then the highest priority of any task which will attempt to obtain this semaphore must be specified. RTEMS allocates a Semaphore Control Block (SMCB) from the SMCB free list. This data structure is used by RTEMS to manage the newly created semaphore. Also, a unique semaphore ID is generated and returned to the calling task.

8.3.2 Obtaining Semaphore IDs

When a semaphore is created, RTEMS generates a unique semaphore ID and assigns it to the created semaphore until it is deleted. The semaphore ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_semaphore_create` directive, the semaphore ID is stored in a user provided location. Second, the semaphore ID may be obtained later using the `rtems_semaphore_ident` directive. The semaphore ID is used by other semaphore manager directives to access this semaphore.

8.3.3 Acquiring a Semaphore

The `rtems_semaphore_obtain` directive is used to acquire the specified semaphore. A simplified version of the `rtems_semaphore_obtain` directive can be described as follows:

```
if semaphore's count is greater than zero
   then decrement semaphore's count
   else wait for release of semaphore

return SUCCESSFUL
```

When the semaphore cannot be immediately acquired, one of the following situations applies:

- By default, the calling task will wait forever to acquire the semaphore.
- Specifying `RTEMS_NO_WAIT` forces an immediate return with an error status code.
- Specifying a timeout limits the interval the task will wait before returning with an error status code.

If the task waits to acquire the semaphore, then it is placed in the semaphore's task wait queue in either FIFO or task priority order. If the task blocked waiting for a binary semaphore using priority inheritance and the task's priority is greater than that of the task currently holding the semaphore, then the holding task will inherit the priority of the blocking task. All tasks waiting on a semaphore are returned an error code when the semaphore is deleted.

When a task successfully obtains a semaphore using priority ceiling and the priority ceiling for this semaphore is greater than that of the holder, then the holder's priority will be elevated.

8.3.4 Releasing a Semaphore

The `rtems_semaphore_release` directive is used to release the specified semaphore. A simplified version of the `rtems_semaphore_release` directive can be described as follows:

```
if no tasks are waiting on this semaphore
   then increment semaphore's count
   else assign semaphore to a waiting task

return SUCCESSFUL
```

If this is the outermost release of a binary semaphore that uses priority inheritance or priority ceiling and the task does not currently hold any other binary semaphores, then the task performing the `rtems_semaphore_release` will have its priority restored to its normal value.

8.3.5 Deleting a Semaphore

The `rtems_semaphore_delete` directive removes a semaphore from the system and frees its control block. A semaphore can be deleted by any local task that knows the semaphore's ID. As a result of this directive, all tasks blocked waiting to acquire the semaphore will be readied and returned a status code which indicates that the semaphore was deleted. Any subsequent references to the semaphore's name and ID are invalid.

8.4 Directives

This section details the semaphore manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

8.4.1 SEMAPHORE_CREATE - Create a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_create(
  rtems_name           name,
  rtems_unsigned32     count,
  rtems_attribute      attribute_set,
  rtems_task_priority  priority_ceiling,
  rtems_id            *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - semaphore created successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_TOO_MANY` - too many semaphores created
`RTEMS_NOT_DEFINED` - invalid attribute set
`RTEMS_INVALID_NUMBER` - invalid starting count for binary semaphore
`RTEMS_MP_NOT_CONFIGURED` - multiprocessing not configured
`RTEMS_TOO_MANY` - too many global objects

DESCRIPTION:

This directive creates a semaphore which resides on the local node. The created semaphore has the user-defined name specified in name and the initial count specified in count. For control and maintenance of the semaphore, RTEMS allocates and initializes a SMCB. The RTEMS-assigned semaphore id is returned in id. This semaphore id is used with other semaphore related directives to access the semaphore.

Specifying PRIORITY in attribute_set causes tasks waiting for a semaphore to be serviced according to task priority. When FIFO is selected, tasks are serviced in First In-First Out order.

NOTES:

This directive will not cause the calling task to be preempted.

The priority inheritance and priority ceiling algorithms are only supported for local, binary semaphores that use the priority task wait queue blocking discipline.

The following semaphore attribute constants are defined by RTEMS:

- `RTEMS_FIFO` - tasks wait by FIFO (default)
- `RTEMS_PRIORITY` - tasks wait by priority
- `RTEMS_BINARY_SEMAPHORE` - restrict values to 0 and 1 (default)

- `RTEMS_COUNTING_SEMAPHORE` - no restriction on values
- `RTEMS_NO_INHERIT_PRIORITY` - do not use priority inheritance (default)
- `RTEMS_INHERIT_PRIORITY` - use priority inheritance
- `RTEMS_PRIORITY_CEILING` - use priority ceiling
- `RTEMS_NO_PRIORITY_CEILING` - do not use priority ceiling (default)
- `RTEMS_LOCAL` - local task (default)
- `RTEMS_GLOBAL` - global task

Semaphores should not be made global unless remote tasks must interact with the created semaphore. This is to avoid the system overhead incurred by the creation of a global semaphore. When a global semaphore is created, the semaphore's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including semaphores, is limited by the maximum_global_objects field in the Configuration Table.

8.4.2 SEMAPHORE_IDENT - Get ID of a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_ident(
  rtems_name        name,
  rtems_unsigned32  node,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - semaphore identified successfully
`RTEMS_INVALID_NAME` - semaphore name not found
`RTEMS_INVALID_NODE` - invalid node id

DESCRIPTION:

This directive obtains the semaphore id associated with the semaphore name. If the semaphore name is not unique, then the semaphore id will match one of the semaphores with that name. However, this semaphore id is not guaranteed to correspond to the desired semaphore. The semaphore id is used by other semaphore related directives to access the semaphore.

NOTES:

This directive will not cause the running task to be preempted.

If node is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If node is a valid node number which does not represent the local node, then only the semaphores exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

8.4.3 SEMAPHORE_DELETE - Delete a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - semaphore deleted successfully
`RTEMS_INVALID_ID` - invalid semaphore id
`RTEMS_ILLEGAL_ON_REMOTE_OBJECT` - cannot delete remote semaphore
`RTEMS_RESOURCE_IN_USE` - binary semaphore is in use

DESCRIPTION:

This directive deletes the semaphore specified by id. All tasks blocked waiting to acquire the semaphore will be readied and returned a status code which indicates that the semaphore was deleted. The SMCB for this semaphore is reclaimed by RTEMS.

NOTES:

The calling task will be preempted if it is enabled by the task's execution mode and a higher priority local task is waiting on the deleted semaphore. The calling task will NOT be preempted if all of the tasks that are waiting on the semaphore are remote tasks.

The calling task does not have to be the task that created the semaphore. Any local task that knows the semaphore id can delete the semaphore.

When a global semaphore is deleted, the semaphore id must be transmitted to every node in the system for deletion from the local copy of the global object table.

The semaphore must reside on the local node, even if the semaphore was created with the `RTEMS_GLOBAL` option.

Proxies, used to represent remote tasks, are reclaimed when the semaphore is deleted.

8.4.4 SEMAPHORE_OBTAIN - Acquire a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_obtain(
  rtems_id         id,
  rtems_unsigned32 option_set,
  rtems_interval   timeout
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - semaphore obtained successfully
`RTEMS_UNSATISFIED` - semaphore not available
`RTEMS_TIMEOUT` - timed out waiting for semaphore
`RTEMS_OBJECT_WAS_DELETED` - semaphore deleted while waiting
`RTEMS_INVALID_ID` - invalid semaphore id

DESCRIPTION:

This directive acquires the semaphore specified by id. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` components of the options parameter indicate whether the calling task wants to wait for the semaphore to become available or return immediately if the semaphore is not currently available. With either `RTEMS_WAIT` or `RTEMS_NO_WAIT`, if the current semaphore count is positive, then it is decremented by one and the semaphore is successfully acquired by returning immediately with a successful return code.

If the calling task chooses to return immediately and the current semaphore count is zero or negative, then a status code is returned indicating that the semaphore is not available. If the calling task chooses to wait for a semaphore and the current semaphore count is zero or negative, then it is decremented by one and the calling task is placed on the semaphore's wait queue and blocked. If the semaphore was created with the `RTEMS_PRIORITY` attribute, then the calling task is inserted into the queue according to its priority. However, if the semaphore was created with the `RTEMS_FIFO` attribute, then the calling task is placed at the rear of the wait queue. If the binary semaphore was created with the `RTEMS_INHERIT_PRIORITY` attribute, then the priority of the task currently holding the binary semaphore is guaranteed to be greater than or equal to that of the blocking task. If the binary semaphore was created with the `RTEMS_PRIORITY_CEILING` attribute, a task successfully obtains the semaphore, and the priority of that task is greater than the ceiling priority for this semaphore, then the priority of the task obtaining the semaphore is elevated to that of the ceiling.

The timeout parameter specifies the maximum interval the calling task is willing to be blocked waiting for the semaphore. If it is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever. If the semaphore is available or the `RTEMS_NO_WAIT` option component is set, then timeout is ignored.

NOTES:

The following semaphore acquisition option constants are defined by RTEMS:

- `RTEMS_WAIT` - task will wait for semaphore (default)
- `RTEMS_NO_WAIT` - task should not wait

Attempting to obtain a global semaphore which does not reside on the local node will generate a request to the remote node to access the semaphore. If the semaphore is not available and `RTEMS_NO_WAIT` was not specified, then the task must be blocked until the semaphore is released. A proxy is allocated on the remote node to represent the task until the semaphore is released.

A clock tick is required to support the timeout functionality of this directive.

8.4.5 SEMAPHORE_RELEASE - Release a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_release(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - semaphore released successfully
`RTEMS_INVALID_ID` - invalid semaphore id
`RTEMS_NOT_OWNER_OF_RESOURCE` - calling task does not own semaphore

DESCRIPTION:

This directive releases the semaphore specified by id. The semaphore count is incremented by one. If the count is zero or negative, then the first task on this semaphore's wait queue is removed and unblocked. The unblocked task may preempt the running task if the running task's preemption mode is enabled and the unblocked task has a higher priority than the running task.

NOTES:

The calling task may be preempted if it causes a higher priority task to be made ready for execution.

Releasing a global semaphore which does not reside on the local node will generate a request telling the remote node to release the semaphore.

If the task to be unblocked resides on a different node from the semaphore, then the semaphore allocation is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

The outermost release of a local, binary, priority inheritance or priority ceiling semaphore may result in the calling task having its priority lowered. This will occur if the calling task holds no other binary semaphores and it has inherited a higher priority.

9 Message Manager

9.1 Introduction

The message manager provides communication and synchronization capabilities using RTEMS message queues. The directives provided by the message manager are:

- `rtems_message_queue_create` - Create a queue
- `rtems_message_queue_ident` - Get ID of a queue
- `rtems_message_queue_delete` - Delete a queue
- `rtems_message_queue_send` - Put message at rear of a queue
- `rtems_message_queue_urgent` - Put message at front of a queue
- `rtems_message_queue_broadcast` - Broadcast N messages to a queue
- `rtems_message_queue_receive` - Receive message from a queue
- `rtems_message_queue_get_number_pending` - Get number of messages pending on a queue
- `rtems_message_queue_flush` - Flush all messages on a queue

9.2 Background

9.2.1 Messages

A message is a variable length buffer where information can be stored to support communication. The length of the message and the information stored in that message are user-defined and can be actual data, pointer(s), or empty.

9.2.2 Message Queues

A message queue permits the passing of messages among tasks and ISRs. Message queues can contain a variable number of messages. Normally messages are sent to and received from the queue in FIFO order using the `rtems_message_queue_send` directive. However, the `rtems_message_queue_urgent` directive can be used to place messages at the head of a queue in LIFO order.

Synchronization can be accomplished when a task can wait for a message to arrive at a queue. Also, a task may poll a queue for the arrival of a message.

The maximum length message which can be sent is set on a per message queue basis.

9.2.3 Building a Message Queue's Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The set of valid message queue attributes is provided in the following table:

- `RTEMS_FIFO` - tasks wait by FIFO (default)
- `RTEMS_PRIORITY` - tasks wait by priority
- `RTEMS_LOCAL` - local message queue (default)
- `RTEMS_GLOBAL` - global message queue

An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute `RTEMS_DEFAULT_ATTRIBUTES` should be specified on this call.

This example demonstrates the attribute_set parameter needed to create a local message queue with the task priority waiting queue discipline. The attribute_set parameter to the `rtems_message_queue_create` directive could be either `RTEMS_PRIORITY` or `RTEMS_LOCAL | RTEMS_PRIORITY`. The attribute_set parameter can be set to `RTEMS_PRIORITY` because `RTEMS_LOCAL` is the default for all created message queues. If a similar message queue were to be known globally, then the attribute_set parameter would be `RTEMS_GLOBAL | RTEMS_PRIORITY`.

9.2.4 Building a MESSAGE_QUEUE_RECEIVE Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_message_queue_receive` directive are listed in the following table:

- `RTEMS_WAIT` - task will wait for a message (default)
- `RTEMS_NO_WAIT` - task should not wait

An option listed as a default is not required to appear in the option OR list, although it is a good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for a message to arrive. The option parameter passed to the `rtems_message_queue_receive` directive should be `RTEMS_NO_WAIT`.

9.3 Operations

9.3.1 Creating a Message Queue

The `rtems_message_queue_create` directive creates a message queue with the user-defined name. The user specifies the maximum message size and maximum number of messages which can be placed in the message queue at one time. The user may select FIFO or task priority as the method for placing waiting tasks in the task wait queue. RTEMS allocates a Queue Control Block (QCB) from the QCB free list to maintain the newly created queue as well as memory for the message buffer pool associated with this message queue. RTEMS also generates a message queue ID which is returned to the calling task.

For GLOBAL message queues, the maximum message size is effectively limited to the longest message which the MPCI is capable of transmitting.

9.3.2 Obtaining Message Queue IDs

When a message queue is created, RTEMS generates a unique message queue ID. The message queue ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_message_queue_create` directive, the queue ID is stored in a user provided location. Second, the queue ID may be obtained later using the `rtems_message_queue_ident` directive. The queue ID is used by other message manager directives to access this message queue.

9.3.3 Receiving a Message

The `rtems_message_queue_receive` directive attempts to retrieve a message from the specified message queue. If at least one message is in the queue, then the message is removed from the queue, copied to the caller's message buffer, and returned immediately along with the length of the message. When messages are unavailable, one of the following situations applies:

- By default, the calling task will wait forever for the message to arrive.
- Specifying the `RTEMS_NO_WAIT` option forces an immediate return with an error status code.
- Specifying a timeout limits the period the task will wait before returning with an error status.

If the task waits for a message, then it is placed in the message queue's task wait queue in either FIFO or task priority order. All tasks waiting on a message queue are returned an error code when the message queue is deleted.

9.3.4 Sending a Message

Messages can be sent to a queue with the `rtems_message_queue_send` and `rtems_message_queue_urgent` directives. These directives work identically when tasks are waiting to receive a message. A task is removed from the task waiting queue, unblocked, and the message is copied to a waiting task's message buffer.

When no tasks are waiting at the queue, `rtems_message_queue_send` places the message at the rear of the message queue, while `rtems_message_queue_urgent` places the message at the front of the queue. The message is copied to a message buffer from this message queue's buffer pool and then placed in the message queue. Neither directive can successfully send a message to a message queue which has a full queue of pending messages.

9.3.5 Broadcasting a Message

The `rtems_message_queue_broadcast` directive sends the same message to every task waiting on the specified message queue as an atomic operation. The message is copied to each waiting task's message buffer and each task is unblocked. The number of tasks which were unblocked is returned to the caller.

9.3.6 Deleting a Message Queue

The `rtems_message_queue_delete` directive removes a message queue from the system and frees its control block as well as the memory associated with this message queue's message buffer pool. A message queue can be deleted by any local task that knows the message queue's ID. As a result of this directive, all tasks blocked waiting to receive a message from the message queue will be readied and returned a status code which indicates that the message queue was deleted. Any subsequent references to the message queue's name and ID are invalid. Any messages waiting at the message queue are also deleted and deallocated.

9.4 Directives

This section details the message manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

9.4.1 MESSAGE_QUEUE_CREATE - Create a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_create(
  rtems_name        name,
  rtems_unsigned32  count,
  rtems_unsigned32  max_message_size,
  rtems_attribute   attribute_set,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - queue created successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_INVALID_NUMBER` - invalid message count
`RTEMS_INVALID_SIZE` - invalid message size
`RTEMS_TOO_MANY` - too many queues created
`RTEMS_MP_NOT_CONFIGURED` - multiprocessing not configured
`RTEMS_TOO_MANY` - too many global objects

DESCRIPTION:

This directive creates a message queue which resides on the local node with the user-defined name specified in name. For control and maintenance of the queue, RTEMS allocates and initializes a QCB. Memory is allocated from the RTEMS Workspace for the specified count of messages, each of max_message_size bytes in length. The RTEMS-assigned queue id, returned in id, is used to access the message queue.

Specifying `RTEMS_PRIORITY` in attribute_set causes tasks waiting for a message to be serviced according to task priority. When `RTEMS_FIFO` is specified, waiting tasks are serviced in First In-First Out order.

NOTES:

This directive will not cause the calling task to be preempted.

The following message queue attribute constants are defined by RTEMS:

- `RTEMS_FIFO` - tasks wait by FIFO (default)
- `RTEMS_PRIORITY` - tasks wait by priority
- `RTEMS_LOCAL` - local message queue (default)
- `RTEMS_GLOBAL` - global message queue

Message queues should not be made global unless remote tasks must interact with the created message queue. This is to avoid the system overhead incurred by the creation of a global message queue. When a global message queue is created, the message queue's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

For GLOBAL message queues, the maximum message size is effectively limited to the longest message which the MPCI is capable of transmitting.

The total number of global objects, including message queues, is limited by the maximum_global_objects field in the configuration table.

9.4.2 MESSAGE_QUEUE_IDENT - Get ID of a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_ident(
  rtems_name        name,
  rtems_unsigned32  node,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - queue identified successfully
`RTEMS_INVALID_NAME` - queue name not found
`RTEMS_INVALID_NODE` - invalid node id

DESCRIPTION:

This directive obtains the queue id associated with the queue name specified in name. If the queue name is not unique, then the queue id will match one of the queues with that name. However, this queue id is not guaranteed to correspond to the desired queue. The queue id is used with other message related directives to access the message queue.

NOTES:

This directive will not cause the running task to be preempted.

If node is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If node is a valid node number which does not represent the local node, then only the message queues exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

9.4.3 MESSAGE_QUEUE_DELETE - Delete a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - queue deleted successfully
`RTEMS_INVALID_ID` - invalid queue id
`RTEMS_ILLEGAL_ON_REMOTE_OBJECT` - cannot delete remote queue

DESCRIPTION:

This directive deletes the message queue specified by id. As a result of this directive, all tasks blocked waiting to receive a message from this queue will be readied and returned a status code which indicates that the message queue was deleted. If no tasks are waiting, but the queue contains messages, then RTEMS returns these message buffers back to the system message buffer pool. The QCB for this queue as well as the memory for the message buffers is reclaimed by RTEMS.

NOTES:

The calling task will be preempted if its preemption mode is enabled and one or more local tasks with a higher priority than the calling task are waiting on the deleted queue. The calling task will NOT be preempted if the tasks that are waiting are remote tasks.

The calling task does not have to be the task that created the queue, although the task and queue must reside on the same node.

When the queue is deleted, any messages in the queue are returned to the free message buffer pool. Any information stored in those messages is lost.

When a global message queue is deleted, the message queue id must be transmitted to every node in the system for deletion from the local copy of the global object table.

Proxies, used to represent remote tasks, are reclaimed when the message queue is deleted.

9.4.4 MESSAGE_QUEUE_SEND - Put message at rear of a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_send(
  rtems_id           id,
  void              *buffer,
  rtems_unsigned32   size
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - message sent successfully
`RTEMS_INVALID_ID` - invalid queue id
`RTEMS_INVALID_SIZE` - invalid message size
`RTEMS_UNSATISFIED` - out of message buffers
`RTEMS_TOO_MANY` - queue's limit has been reached

DESCRIPTION:

This directive sends the message buffer of size bytes in length to the queue specified by id. If a task is waiting at the queue, then the message is copied to the waiting task's buffer and the task is unblocked. If no tasks are waiting at the queue, then the message is copied to a message buffer which is obtained from this message queue's message buffer pool. The message buffer is then placed at the rear of the queue.

NOTES:

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

Sending a message to a global message queue which does not reside on the local node will generate a request to the remote node to post the message on the specified message queue.

If the task to be unblocked resides on a different node from the message queue, then the message is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

9.4.5 MESSAGE_QUEUE_URGENT - Put message at front of a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_urgent(
  rtems_id           id,
  void              *buffer,
  rtems_unsigned32   size
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - message sent successfully
`RTEMS_INVALID_ID` - invalid queue id
`RTEMS_INVALID_SIZE` - invalid message size
`RTEMS_UNSATISFIED` - out of message buffers
`RTEMS_TOO_MANY` - queue's limit has been reached

DESCRIPTION:

This directive sends the message buffer of size bytes in length to the queue specified by id. If a task is waiting on the queue, then the message is copied to the task's buffer and the task is unblocked. If no tasks are waiting on the queue, then the message is copied to a message buffer which is obtained from this message queue's message buffer pool. The message buffer is then placed at the front of the queue.

NOTES:

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

Sending a message to a global message queue which does not reside on the local node will generate a request telling the remote node to post the message on the specified message queue.

If the task to be unblocked resides on a different node from the message queue, then the message is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

9.4.6 MESSAGE_QUEUE_BROADCAST - Broadcast N messages to a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_broadcast(
  rtems_id           id,
  void              *buffer,
  rtems_unsigned32   size,
  rtems_unsigned32  *count
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - message broadcasted successfully
`RTEMS_INVALID_ID` - invalid queue id
`RTEMS_INVALID_SIZE` - invalid message size

DESCRIPTION:

This directive causes all tasks that are waiting at the queue specified by id to be unblocked and sent the message contained in buffer. Before a task is unblocked, the message buffer of size byes in length is copied to that task's message buffer. The number of tasks that were unblocked is returned in count.

NOTES:

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

The execution time of this directive is directly related to the number of tasks waiting on the message queue, although it is more efficient than the equivalent number of invocations of `rtems_message_queue_send`.

Broadcasting a message to a global message queue which does not reside on the local node will generate a request telling the remote node to broadcast the message to the specified message queue.

When a task is unblocked which resides on a different node from the message queue, a copy of the message is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

9.4.7 MESSAGE_QUEUE_RECEIVE - Receive message from a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_receive(
  rtems_id           id,
  void              *buffer,
  rtems_unsigned32  *size,
  rtems_unsigned32   option_set,
  rtems_interval     timeout
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - message received successfully
`RTEMS_INVALID_ID` - invalid queue id
`RTEMS_UNSATISFIED` - queue is empty
`RTEMS_TIMEOUT` - timed out waiting for message
`RTEMS_OBJECT_WAS_DELETED` - queue deleted while waiting

DESCRIPTION:

This directive receives a message from the message queue specified in id. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` options of the options parameter allow the calling task to specify whether to wait for a message to become available or return immediately. For either option, if there is at least one message in the queue, then it is copied to buffer, size is set to return the length of the message in bytes, and this directive returns immediately with a successful return code.

If the calling task chooses to return immediately and the queue is empty, then a status code indicating this condition is returned. If the calling task chooses to wait at the message queue and the queue is empty, then the calling task is placed on the message wait queue and blocked. If the queue was created with the `RTEMS_PRIORITY` option specified, then the calling task is inserted into the wait queue according to its priority. But, if the queue was created with the `RTEMS_FIFO` option specified, then the calling task is placed at the rear of the wait queue.

A task choosing to wait at the queue can optionally specify a timeout value in the timeout parameter. The timeout parameter specifies the maximum interval to wait before the calling task desires to be unblocked. If it is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever.

NOTES:

The following message receive option constants are defined by RTEMS:

- `RTEMS_WAIT` - task will wait for a message (default)
- `RTEMS_NO_WAIT` - task should not wait

Receiving a message from a global message queue which does not reside on the local node will generate a request to the remote node to obtain a message from the specified message queue. If no message is available and `RTEMS_WAIT` was specified, then the task must be blocked until a message is posted. A proxy is allocated on the remote node to represent the task until the message is posted.

A clock tick is required to support the timeout functionality of this directive.

9.4.8 MESSAGE_QUEUE_GET_NUMBER_PENDING - Get number of messages pending on a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_get_number_pending(
  rtems_id          id,
  rtems_unsigned32 *count
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - number of messages pending returned successfully
`RTEMS_INVALID_ID` - invalid queue id

DESCRIPTION:

This directive returns the number of messages pending on this message queue in count. If no messages are present on the queue, count is set to zero.

NOTES:

Getting the number of pending messages on a global message queue which does not reside on the local node will generate a request to the remote node to actually obtain the pending message count for the specified message queue.

9.4.9 MESSAGE_QUEUE_FLUSH - Flush all messages on a queue

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_flush(
  rtems_id          id,
  rtems_unsigned32 *count
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - message queue flushed successfully
`RTEMS_INVALID_ID` - invalid queue id

DESCRIPTION:

This directive removes all pending messages from the specified queue id. The number of messages removed is returned in count. If no messages are present on the queue, count is set to zero.

NOTES:

Flushing all messages on a global message queue which does not reside on the local node will generate a request to the remote node to actually flush the specified message queue.

10 Event Manager

10.1 Introduction

The event manager provides a high performance method of intertask communication and synchronization. The directives provided by the event manager are:

- `rtems_event_send` - Send event set to a task
- `rtems_event_receive` - Receive event condition

10.2 Background

10.2.1 Event Sets

An event flag is used by a task (or ISR) to inform another task of the occurrence of a significant situation. Thirty-two event flags are associated with each task. A collection of one or more event flags is referred to as an event set. The application developer should remember the following key characteristics of event operations when utilizing the event manager:

- Events provide a simple synchronization facility.
- Events are aimed at tasks.
- Tasks can wait on more than one event simultaneously.
- Events are independent of one another.
- Events do not hold or transport data.
- Events are not queued. In other words, if an event is sent more than once to a task before being received, the second and subsequent send operations to that same task have no effect.

An event set is posted when it is directed (or sent) to a task. A pending event is an event that has been posted but not received. An event condition is used to specify the events which the task desires to receive and the algorithm which will be used to determine when the request is satisfied. An event condition is satisfied based upon one of two algorithms which are selected by the user. The `RTEMS_EVENT_ANY` algorithm states that an event condition is satisfied when at least a single requested event is posted. The `RTEMS_EVENT_ALL` algorithm states that an event condition is satisfied when every requested event is posted.

10.2.2 Building an Event Set or Condition

An event set or condition is built by a bitwise OR of the desired events. The set of valid events is `RTEMS_EVENT_0` through `RTEMS_EVENT_31`. If an event is not explicitly specified in the set or condition, then it is not present. Events are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each event appears exactly once in the event set list.

For example, when sending the event set consisting of `RTEMS_EVENT_6`, `RTEMS_EVENT_15`, and `RTEMS_EVENT_31`, the event parameter to the `rtems_event_send` directive should be `RTEMS_EVENT_6 | RTEMS_EVENT_15 | RTEMS_EVENT_31`.

10.2.3 Building an EVENT_RECEIVE Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_event_receive` directive are listed in the following table:

- `RTEMS_WAIT` - task will wait for event (default)
- `RTEMS_NO_WAIT` - task should not wait
- `RTEMS_EVENT_ALL` - return after all events (default)
- `RTEMS_EVENT_ANY` - return after any events

Option values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each option appears exactly once in the component list. An option listed as a default is not required to appear in the option list, although it is a good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for all events in a particular event condition to arrive. The option parameter passed to the `rtems_event_receive` directive should be either `RTEMS_EVENT_ALL | RTEMS_NO_WAIT` or `RTEMS_NO_WAIT`. The option parameter can be set to `RTEMS_NO_WAIT` because `RTEMS_EVENT_ALL` is the default condition for `rtems_event_receive`.

10.3 Operations

10.3.1 Sending an Event Set

The `rtems_event_send` directive allows a task (or an ISR) to direct an event set to a target task. Based upon the state of the target task, one of the following situations applies:

- Target Task is Blocked Waiting for Events
    - If the waiting task's input event condition is satisfied, then the task is made ready for execution.
    - If the waiting task's input event condition is not satisfied, then the event set is posted but left pending and the task remains blocked.
- Target Task is Not Waiting for Events
    - The event set is posted and left pending.

10.3.2 Receiving an Event Set

The `rtems_event_receive` directive is used by tasks to accept a specific input event condition. The task also specifies whether the request is satisfied when all requested events are available or any single requested event is available. If the requested event condition is satisfied by pending events, then a successful return code and the satisfying event set are returned immediately. If the condition is not satisfied, then one of the following situations applies:

- By default, the calling task will wait forever for the event condition to be satisfied.
- Specifying the `RTEMS_NO_WAIT` option forces an immediate return with an error status code.
- Specifying a timeout limits the period the task will wait before returning with an error status code.

10.3.3 Determining the Pending Event Set

A task can determine the pending event set by calling the `rtems_event_receive` directive with a value of `RTEMS_PENDING_EVENTS` for the input event condition. The pending events are returned to the calling task but the event set is left unaltered.

10.3.4 Receiving all Pending Events

A task can receive all of the currently pending events by calling the `rtems_event_receive` directive with a value of `RTEMS_ALL_EVENTS` for the input event condition and `RTEMS_NO_WAIT | RTEMS_EVENT_ANY` for the option set. The pending events are returned to the calling task and the event set is cleared. If no events are pending then the `RTEMS_UNSATISFIED` status code will be returned.

10.4 Directives

This section details the event manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

10.4.1 EVENT_SEND - Send event set to a task

CALLING SEQUENCE:

```
rtems_status_code rtems_event_send (
  rtems_id         id,
  rtems_event_set  event_in
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - event set sent successfully
`RTEMS_INVALID_ID` - invalid task id

DESCRIPTION:

This directive sends an event set, event_in, to the task specified by id. If a blocked task's input event condition is satisfied by this directive, then it will be made ready. If its input event condition is not satisfied, then the events satisfied are updated and the events not satisfied are left pending. If the task specified by id is not blocked waiting for events, then the events sent are left pending.

NOTES:

Specifying `RTEMS_SELF` for id results in the event set being sent to the calling task.

Identical events sent to a task are not queued. In other words, the second, and subsequent, posting of an event to a task before it can perform an `rtems_event_receive` has no effect.

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

Sending an event set to a global task which does not reside on the local node will generate a request telling the remote node to send the event set to the appropriate task.

10.4.2 EVENT_RECEIVE - Receive event condition

CALLING SEQUENCE:

```
rtems_status_code rtems_event_receive (
  rtems_event_set  event_in,
  rtems_option     option_set,
  rtems_interval   ticks,
  rtems_event_set *event_out
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - event received successfully
`RTEMS_UNSATISFIED` - input event not satisfied (`RTEMS_NO_WAIT`)
`RTEMS_TIMEOUT` - timed out waiting for event

DESCRIPTION:

This directive attempts to receive the event condition specified in event_in. If event_in is set to `RTEMS_PENDING_EVENTS`, then the current pending events are returned in event_out and left pending. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` options in the option_set parameter are used to specify whether or not the task is willing to wait for the event condition to be satisfied. `RTEMS_EVENT_ANY` and `RTEMS_EVENT_ALL` are used in the option_set parameter are used to specify whether a single event or the complete event set is necessary to satisfy the event condition. The event_out parameter is returned to the calling task with the value that corresponds to the events in event_in that were satisfied.

If pending events satisfy the event condition, then event_out is set to the satisfied events and the pending events in the event condition are cleared. If the event condition is not satisfied and `RTEMS_NO_WAIT` is specified, then event_out is set to the currently satisfied events. If the calling task chooses to wait, then it will block waiting for the event condition.

If the calling task must wait for the event condition to be satisfied, then the timeout parameter is used to specify the maximum interval to wait. If it is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever.

NOTES:

This directive only affects the events specified in event_in. Any pending events that do not correspond to any of the events specified in event_in will be left pending.

The following event receive option constants are defined by RTEMS:

- `RTEMS_WAIT` task will wait for event (default)

- `RTEMS_NO_WAIT` task should not wait
- `RTEMS_EVENT_ALL` return after all events (default)
- `RTEMS_EVENT_ANY` return after any events

A clock tick is required to support the functionality of this directive.

11 Signal Manager

11.1 Introduction

The signal manager provides the capabilities required for asynchronous communication. The directives provided by the signal manager are:

- `rtems_signal_catch` - Establish an ASR
- `rtems_signal_send` - Send signal set to a task

11.2 Background

11.2.1 Signal Manager Definitions

The signal manager allows a task to optionally define an asynchronous signal routine (ASR). An ASR is to a task what an ISR is to an application's set of tasks. When the processor is interrupted, the execution of an application is also interrupted and an ISR is given control. Similarly, when a signal is sent to a task, that task's execution path will be "interrupted" by the ASR. Sending a signal to a task has no effect on the receiving task's current execution state.

A signal flag is used by a task (or ISR) to inform another task of the occurrence of a significant situation. Thirty-two signal flags are associated with each task. A collection of one or more signals is referred to as a signal set. A signal set is posted when it is directed (or sent) to a task. A pending signal is a signal that has been sent to a task with a valid ASR, but has not been processed by that task's ASR.

11.2.2 A Comparison of ASRs and ISRs

The format of an ASR is similar to that of an ISR with the following exceptions:

- ISRs are scheduled by the processor hardware. ASRs are scheduled by RTEMS.
- ISRs do not execute in the context of a task and may invoke only a subset of directives. ASRs execute in the context of a task and may execute any directive.
- When an ISR is invoked, it is passed the vector number as its argument. When an ASR is invoked, it is passed the signal set as its argument.
- An ASR has a task mode which can be different from that of the task. An ISR does not execute as a task and, as a result, does not have a task mode.

11.2.3 Building a Signal Set

A signal set is built by a bitwise OR of the desired signals. The set of valid signals is `RTEMS_SIGNAL_0` through `RTEMS_SIGNAL_31`. If a signal is not explicitly specified in the signal set, then it is not present. Signal values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each signal appears exactly once in the component list.

This example demonstrates the signal parameter used when sending the signal set consisting of `RTEMS_SIGNAL_6`, `RTEMS_SIGNAL_15`, and `RTEMS_SIGNAL_31`. The signal parameter provided to the `rtems_signal_send` directive should be `RTEMS_SIGNAL_6 | RTEMS_SIGNAL_15 | RTEMS_SIGNAL_31`.

11.2.4 Building an ASR's Mode

In general, an ASR's mode is built by a bitwise OR of the desired mode components. The set of valid mode components is the same as those allowed with the task_create and task_mode directives. A complete list of mode options is provided in the following table:

- `RTEMS_PREEMPT` is masked by `RTEMS_PREEMPT_MASK` and enables preemption
- `RTEMS_NO_PREEMPT` is masked by `RTEMS_PREEMPT_MASK` and disables preemption
- `RTEMS_NO_TIMESLICE` is masked by `RTEMS_TIMESLICE_MASK` and disables timeslicing
- `RTEMS_TIMESLICE` is masked by `RTEMS_TIMESLICE_MASK` and enables timeslicing
- `RTEMS_ASR` is masked by `RTEMS_ASR_MASK` and enables ASR processing
- `RTEMS_NO_ASR` is masked by `RTEMS_ASR_MASK` and disables ASR processing
- `RTEMS_INTERRUPT_LEVEL(0)` is masked by `RTEMS_INTERRUPT_MASK` and enables all interrupts
- `RTEMS_INTERRUPT_LEVEL(n)` is masked by `RTEMS_INTERRUPT_MASK` and sets interrupts level n

Mode values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each mode appears exactly once in the component list. A mode component listed as a default is not required to appear in the mode list, although it is a good programming practice to specify default components. If all defaults are desired, the mode DEFAULT_MODES should be specified on this call.

This example demonstrates the mode parameter used with the `rtems_signal_catch` to establish an ASR which executes at interrupt level three and is non-preemptible. The mode should be set to `RTEMS_INTERRUPT_LEVEL(3) | RTEMS_NO_PREEMPT` to indicate the desired processor mode and interrupt level.

11.3 Operations

11.3.1 Establishing an ASR

The `rtems_signal_catch` directive establishes an ASR for the calling task. The address of the ASR and its execution mode are specified to this directive. The ASR's mode is distinct from the task's mode. For example, the task may allow preemption, while that task's ASR may have preemption disabled. Until a task calls `rtems_signal_catch` the first time, its ASR is invalid, and no signal sets can be sent to the task.

A task may invalidate its ASR and discard all pending signals by calling `rtems_signal_catch` with a value of NULL for the ASR's address. When a task's ASR is invalid, new signal sets sent to this task are discarded.

A task may disable ASR processing (`RTEMS_NO_ASR`) via the task_mode directive. When a task's ASR is disabled, the signals sent to it are left pending to be processed later when the ASR is enabled.

Any directive that can be called from a task can also be called from an ASR. A task is only allowed one active ASR. Thus, each call to `rtems_signal_catch` replaces the previous one.

Normally, signal processing is disabled for the ASR's execution mode, but if signal processing is enabled for the ASR, the ASR must be reentrant.

11.3.2 Sending a Signal Set

The `rtems_signal_send` directive allows both tasks and ISRs to send signals to a target task. The target task and a set of signals are specified to the `rtems_signal_send` directive. The sending of a signal to a task has no effect on the execution state of that task. If the task is not the currently running task, then the signals are left pending and processed by the task's ASR the next time the task is dispatched to run. The ASR is executed immediately before the task is dispatched. If the currently running task sends a signal to itself or is sent a signal from an ISR, its ASR is immediately dispatched to run provided signal processing is enabled.

If an ASR with signals enabled is preempted by another task or an ISR and a new signal set is sent, then a new copy of the ASR will be invoked, nesting the preempted ASR. Upon completion of processing the new signal set, control will return to the preempted ASR. In this situation, the ASR must be reentrant.

Like events, identical signals sent to a task are not queued. In other words, sending the same signal multiple times to a task (without any intermediate signal processing occurring for the task), has the same result as sending that signal to that task once.

11.3.3 Processing an ASR

Asynchronous signals were designed to provide the capability to generate software interrupts. The processing of software interrupts parallels that of hardware interrupts. As a result, the differences between the formats of ASRs and ISRs is limited to the meaning of the single argument passed to an ASR. The ASR should have the following calling sequence and adhere to C calling conventions:

```
rtems_asr user_routine(
  rtems_signal_set signals
);
```

When the ASR returns to RTEMS the mode and execution path of the interrupted task (or ASR) is restored to the context prior to entering the ASR.

11.4 Directives

This section details the signal manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

11.4.1 SIGNAL_CATCH - Establish an ASR

CALLING SEQUENCE:

```
rtems_status_code rtems_signal_catch(
  rtems_asr_entry  asr_handler,
  rtems_mode mode
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - always successful

DESCRIPTION:

This directive establishes an asynchronous signal routine (ASR) for the calling task. The asr_handler parameter specifies the entry point of the ASR. If asr_handler is NULL, the ASR for the calling task is invalidated and all pending signals are cleared. Any signals sent to a task with an invalid ASR are discarded. The mode parameter specifies the execution mode for the ASR. This execution mode supersedes the task's execution mode while the ASR is executing.

NOTES:

This directive will not cause the calling task to be preempted.

The following task mode constants are defined by RTEMS:

- `RTEMS_PREEMPT` is masked by `RTEMS_PREEMPT_MASK` and enables preemption
- `RTEMS_NO_PREEMPT` is masked by `RTEMS_PREEMPT_MASK` and disables preemption
- `RTEMS_NO_TIMESLICE` is masked by `RTEMS_TIMESLICE_MASK` and disables timeslicing
- `RTEMS_TIMESLICE` is masked by `RTEMS_TIMESLICE_MASK` and enables timeslicing
- `RTEMS_ASR` is masked by `RTEMS_ASR_MASK` and enables ASR processing
- `RTEMS_NO_ASR` is masked by `RTEMS_ASR_MASK` and disables ASR processing
- `RTEMS_INTERRUPT_LEVEL(0)` is masked by `RTEMS_INTERRUPT_MASK` and enables all interrupts
- `RTEMS_INTERRUPT_LEVEL(n)` is masked by `RTEMS_INTERRUPT_MASK` and sets interrupts level n

11.4.2 SIGNAL_SEND - Send signal set to a task

CALLING SEQUENCE:

```
rtems_status_code rtems_signal_send(
  rtems_id         id,
  rtems_signal_set signal_set
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - signal sent successfully
`RTEMS_INVALID_ID` - task id invalid
`RTEMS_NOT_DEFINED` - ASR invalid

DESCRIPTION:

This directive sends a signal set to the task specified in id. The signal_set parameter contains the signal set to be sent to the task.

If a caller sends a signal set to a task with an invalid ASR, then an error code is returned to the caller. If a caller sends a signal set to a task whose ASR is valid but disabled, then the signal set will be caught and left pending for the ASR to process when it is enabled. If a caller sends a signal set to a task with an ASR that is both valid and enabled, then the signal set is caught and the ASR will execute the next time the task is dispatched to run.

NOTES:

Sending a signal set to a task has no effect on that task's state. If a signal set is sent to a blocked task, then the task will remain blocked and the signals will be processed when the task becomes the running task.

Sending a signal set to a global task which does not reside on the local node will generate a request telling the remote node to send the signal set to the specified task.

12 Partition Manager

12.1 Introduction

The partition manager provides facilities to dynamically allocate memory in fixed-size units. The directives provided by the partition manager are:

- `rtems_partition_create` - Create a partition
- `rtems_partition_ident` - Get ID of a partition
- `rtems_partition_delete` - Delete a partition
- `rtems_partition_get_buffer` - Get buffer from a partition
- `rtems_partition_return_buffer` - Return buffer to a partition

12.2 Background

12.2.1 Partition Manager Definitions

A partition is a physically contiguous memory area divided into fixed-size buffers that can be dynamically allocated and deallocated.

Partitions are managed and maintained as a list of buffers. Buffers are obtained from the front of the partition's free buffer chain and returned to the rear of the same chain. When a buffer is on the free buffer chain, RTEMS uses eight bytes of each buffer as the free buffer chain. When a buffer is allocated, the entire buffer is available for application use. Therefore, modifying memory that is outside of an allocated buffer could destroy the free buffer chain or the contents of an adjacent allocated buffer.

12.2.2 Building a Partition's Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The set of valid partition attributes is provided in the following table:

- `RTEMS_LOCAL` - local task (default)
- `RTEMS_GLOBAL` - global task

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute `RTEMS_DEFAULT_ATTRIBUTES` should be specified on this call. The attribute_set parameter should be `RTEMS_GLOBAL` to indicate that the partition is to be known globally.

12.3 Operations

12.3.1 Creating a Partition

The `rtems_partition_create` directive creates a partition with a user-specified name. The partition's name, starting address, length and buffer size are all specified to the `rtems_partition_create` directive. RTEMS allocates a Partition Control Block (PTCB) from the PTCB free list. This data structure is used by RTEMS to manage the newly created partition. The number of buffers in the partition is calculated based upon the specified partition length and buffer size, and returned to the calling task along with a unique partition ID.

12.3.2 Obtaining Partition IDs

When a partition is created, RTEMS generates a unique partition ID and assigned it to the created partition until it is deleted. The partition ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_partition_create` directive, the partition ID is stored in a user provided location. Second, the partition ID may be obtained later using the `rtems_partition_ident` directive. The partition ID is used by other partition manager directives to access this partition.

12.3.3 Acquiring a Buffer

A buffer can be obtained by calling the `rtems_partition_get_buffer` directive. If a buffer is available, then it is returned immediately with a successful return code. Otherwise, an unsuccessful return code is returned immediately to the caller. Tasks cannot block to wait for a buffer to become available.

12.3.4 Releasing a Buffer

Buffers are returned to a partition's free buffer chain with the `rtems_partition_return_buffer` directive. This directive returns an error status code if the returned buffer was not previously allocated from this partition.

12.3.5 Deleting a Partition

The `rtems_partition_delete` directive allows a partition to be removed and returned to RTEMS. When a partition is deleted, the PTCB for that partition is returned to the PTCB free list. A partition with buffers still allocated cannot be deleted. Any task attempting to do so will be returned an error status code.

12.4 Directives

This section details the partition manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

12.4.1 PARTITION_CREATE - Create a partition

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_create(
  rtems_name        name,
  void             *starting_address,
  rtems_unsigned32  length,
  rtems_unsigned32  buffer_size,
  rtems_attribute   attribute_set,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - partition created successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_TOO_MANY` - too many partitions created
`RTEMS_INVALID_ADDRESS` - address not on four byte boundary
`RTEMS_INVALID_SIZE` - length or buffer size is 0
`RTEMS_INVALID_SIZE` - length is less than the buffer size
`RTEMS_INVALID_SIZE` - buffer size not a multiple of 4
`RTEMS_MP_NOT_CONFIGURED` - multiprocessing not configured
`RTEMS_TOO_MANY` - too many global objects

DESCRIPTION:

This directive creates a partition of fixed size buffers from a physically contiguous memory space which starts at starting_address and is length bytes in size. Each allocated buffer is to be of buffer_length in bytes. The assigned partition id is returned in id. This partition id is used to access the partition with other partition related directives. For control and maintenance of the partition, RTEMS allocates a PTCB from the local PTCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

The starting_address and buffer_size parameters must be multiples of four.

Memory from the partition is not used by RTEMS to store the Partition Control Block.

The following partition attribute constants are defined by RTEMS:

- `RTEMS_LOCAL` - local task (default)
- `RTEMS_GLOBAL` - global task

The PTCB for a global partition is allocated on the local node. The memory space used for the partition must reside in shared memory. Partitions should not be made global unless remote tasks must interact with the partition. This is to avoid the overhead incurred by the creation of a global partition. When a global partition is created, the partition's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including partitions, is limited by the maximum_global_objects field in the Configuration Table.

12.4.2 PARTITION_IDENT - Get ID of a partition

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_ident(
  rtems_name        name,
  rtems_unsigned32  node,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - partition identified successfully
`RTEMS_INVALID_NAME` - partition name not found
`RTEMS_INVALID_NODE` - invalid node id

DESCRIPTION:

This directive obtains the partition id associated with the partition name. If the partition name is not unique, then the partition id will match one of the partitions with that name. However, this partition id is not guaranteed to correspond to the desired partition. The partition id is used with other partition related directives to access the partition.

NOTES:

This directive will not cause the running task to be preempted.

If node is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If node is a valid node number which does not represent the local node, then only the partitions exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

12.4.3 PARTITION_DELETE - Delete a partition

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - partition deleted successfully
`RTEMS_INVALID_ID` - invalid partition id
`RTEMS_RESOURCE_IN_USE` - buffers still in use
`RTEMS_ILLEGAL_ON_REMOTE_OBJECT` - cannot delete remote partition

DESCRIPTION:

This directive deletes the partition specified by id. The partition cannot be deleted if any of its buffers are still allocated. The PTCB for the deleted partition is reclaimed by RTEMS.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the partition. Any local task that knows the partition id can delete the partition.

When a global partition is deleted, the partition id must be transmitted to every node in the system for deletion from the local copy of the global object table.

The partition must reside on the local node, even if the partition was created with the `RTEMS_GLOBAL` option.

12.4.4 PARTITION_GET_BUFFER - Get buffer from a partition

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_get_buffer(
  rtems_id   id,
  void     **buffer
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - buffer obtained successfully
`RTEMS_INVALID_ID` - invalid partition id
`RTEMS_UNSATISFIED` - all buffers are allocated

DESCRIPTION:

This directive allows a buffer to be obtained from the partition specified in id. The address of the allocated buffer is returned in buffer.

NOTES:

This directive will not cause the running task to be preempted.

All buffers begin on a four byte boundary.

A task cannot wait on a buffer to become available.

Getting a buffer from a global partition which does not reside on the local node will generate a request telling the remote node to allocate a buffer from the specified partition.

12.4.5 PARTITION_RETURN_BUFFER - Return buffer to a partition

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_return_buffer(
  rtems_id  id,
  void     *buffer
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - buffer returned successfully
`RTEMS_INVALID_ID` - invalid partition id
`RTEMS_INVALID_ADDRESS` - buffer address not in partition

DESCRIPTION:

This directive returns the buffer specified by buffer to the partition specified by id.

NOTES:

This directive will not cause the running task to be preempted.

Returning a buffer to a global partition which does not reside on the local node will generate a request telling the remote node to return the buffer to the specified partition.

13 Region Manager

13.1 Introduction

The region manager provides facilities to dynamically allocate memory in variable sized units. The directives provided by the region manager are:

- `rtems_region_create` - Create a region
- `rtems_region_ident` - Get ID of a region
- `rtems_region_delete` - Delete a region
- `rtems_region_extend` - Add memory to a region
- `rtems_region_get_segment` - Get segment from a region
- `rtems_region_return_segment` - Return segment to a region
- `rtems_region_get_segment_size` - Obtain size of a segment

13.2 Background

13.2.1 Region Manager Definitions

A region makes up a physically contiguous memory space with user-defined boundaries from which variable-sized segments are dynamically allocated and deallocated. A segment is a variable size section of memory which is allocated in multiples of a user-defined page size. This page size is required to be a multiple of four greater than or equal to four. For example, if a request for a 350-byte segment is made in a region with 256-byte pages, then a 512-byte segment is allocated.

Regions are organized as doubly linked chains of variable sized memory blocks. Memory requests are allocated using a first-fit algorithm. If available, the requester receives the number of bytes requested (rounded up to the next page size). RTEMS requires some overhead from the region's memory for each segment that is allocated. Therefore, an application should only modify the memory of a segment that has been obtained from the region. The application should NOT modify the memory outside of any obtained segments and within the region's boundaries while the region is currently active in the system.

Upon return to the region, the free block is coalesced with its neighbors (if free) on both sides to produce the largest possible unused block.

13.2.2 Building an Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The set of valid region attributes is provided in the following table:

- `RTEMS_FIFO` - tasks wait by FIFO (default)

- `RTEMS_PRIORITY` - tasks wait by priority

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute `RTEMS_DEFAULT_ATTRIBUTES` should be specified on this call.

This example demonstrates the attribute_set parameter needed to create a region with the task priority waiting queue discipline. The attribute_set parameter to the `rtems_region_create` directive should be `RTEMS_PRIORITY`.

13.2.3 Building an Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_region_get_segment` directive are listed in the following table:

- `RTEMS_WAIT` - task will wait for semaphore (default)
- `RTEMS_NO_WAIT` - task should not wait

Option values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each option appears exactly once in the component list. An option listed as a default is not required to appear in the option list, although it is a good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for a segment. The option parameter passed to the `rtems_region_get_segment` directive should be `RTEMS_NO_WAIT`.

13.3 Operations

13.3.1 Creating a Region

The `rtems_region_create` directive creates a region with the user-defined name. The user may select FIFO or task priority as the method for placing waiting tasks in the task wait queue. RTEMS allocates a Region Control Block (RNCB) from the RNCB free list to maintain the newly created region. RTEMS also generates a unique region ID which is returned to the calling task.

It is not possible to calculate the exact number of bytes available to the user since RTEMS requires overhead for each segment allocated. For example, a region with one segment that is the size of the entire region has more available bytes than a region with two segments that collectively are the size of the entire region. This is because the region with one segment requires only the overhead for one segment, while the other region requires the overhead for two segments.

Due to automatic coalescing, the number of segments in the region dynamically changes. Therefore, the total overhead required by RTEMS dynamically changes.

13.3.2 Obtaining Region IDs

When a region is created, RTEMS generates a unique region ID and assigns it to the created region until it is deleted. The region ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_region_create` directive, the region ID is stored in a user provided location. Second, the region ID may be obtained later using the `rtems_region_ident` directive. The region ID is used by other region manager directives to access this region.

13.3.3 Adding Memory to a Region

The `rtems_region_extend` directive may be used to add memory to an existing region. The caller specifies the size in bytes and starting address of the memory being added.

NOTE: Please see the release notes or RTEMS source code for information regarding restrictions on the location of the memory being added in relation to memory already in the region.

13.3.4 Acquiring a Segment

The `rtems_region_get_segment` directive attempts to acquire a segment from a specified region. If the region has enough available free memory, then a segment is returned successfully to the caller. When the segment cannot be allocated, one of the following situations applies:

- By default, the calling task will wait forever to acquire the segment.
- Specifying the `RTEMS_NO_WAIT` option forces an immediate return with an error status code.
- Specifying a timeout limits the interval the task will wait before returning with an error status code.

If the task waits for the segment, then it is placed in the region's task wait queue in either FIFO or task priority order. All tasks waiting on a region are returned an error when the message queue is deleted.

13.3.5 Releasing a Segment

When a segment is returned to a region by the `rtems_region_return_segment` directive, it is merged with its unallocated neighbors to form the largest possible segment. The first task on the wait queue is examined to determine if its segment request can now be satisfied. If so, it is given a segment and unblocked. This process is repeated until the first task's segment request cannot be satisfied.

13.3.6 Obtaining the Size of a Segment

The `rtems_region_get_segment_size` directive returns the size in bytes of the specified segment. The size returned includes any "extra" memory included in the segment because of rounding up to a page size boundary.

13.3.7 Deleting a Region

A region can be removed from the system and returned to RTEMS with the `rtems_region_delete` directive. When a region is deleted, its control block is returned to the RNCB free list. A region with segments still allocated is not allowed to be deleted. Any task attempting to do so will be returned an error. As a result of this directive, all tasks blocked waiting to obtain a segment from the region will be readied and returned a status code which indicates that the region was deleted.

13.4 Directives

This section details the region manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

13.4.1 REGION_CREATE - Create a region

CALLING SEQUENCE:

```
rtems_status_code rtems_region_create(
  rtems_name        name,
  void             *starting_address,
  rtems_unsigned32  length,
  rtems_unsigned32  page_size,
  rtems_attribute   attribute_set,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - region created successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_INVALID_ADDRESS` - address not on four byte boundary
`RTEMS_TOO_MANY` - too many regions created
`RTEMS_INVALID_SIZE` - invalid page size

DESCRIPTION:

This directive creates a region from a physically contiguous memory space which starts at starting_address and is length bytes long. Segments allocated from the region will be a multiple of page_size bytes in length. The assigned region id is returned in id. This region id is used as an argument to other region related directives to access the region.

For control and maintenance of the region, RTEMS allocates and initializes an RNCB from the RNCB free pool. Thus memory from the region is not used to store the RNCB. However, some overhead within the region is required by RTEMS each time a segment is constructed in the region.

Specifying `RTEMS_PRIORITY` in attribute_set causes tasks waiting for a segment to be serviced according to task priority. Specifying `RTEMS_FIFO` in attribute_set or selecting `RTEMS_DEFAULT_ATTRIBUTES` will cause waiting tasks to be serviced in First In-First Out order.

The starting_address parameter must be aligned on a four byte boundary. The page_size parameter must be a multiple of four greater than or equal to four.

NOTES:

This directive will not cause the calling task to be preempted.

The following region attribute constants are defined by RTEMS:

- `RTEMS_FIFO` - tasks wait by FIFO (default)

- `RTEMS_PRIORITY` - tasks wait by priority

13.4.2 REGION_IDENT - Get ID of a region

CALLING SEQUENCE:

```
rtems_status_code rtems_region_ident(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - region identified successfully
`RTEMS_INVALID_NAME` - region name not found

DESCRIPTION:

This directive obtains the region id associated with the region name to be acquired. If the region name is not unique, then the region id will match one of the regions with that name. However, this region id is not guaranteed to correspond to the desired region. The region id is used to access this region in other region manager directives.

NOTES:

This directive will not cause the running task to be preempted.

13.4.3 REGION_DELETE - Delete a region

CALLING SEQUENCE:

```
rtems_status_code rtems_region_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - region deleted successfully
`RTEMS_INVALID_ID` - invalid region id
`RTEMS_RESOURCE_IN_USE` - segments still in use

DESCRIPTION:

This directive deletes the region specified by id. The region cannot be deleted if any of its segments are still allocated. The RNCB for the deleted region is reclaimed by RTEMS.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the region. Any local task that knows the region id can delete the region.

13.4.4 REGION_EXTEND - Add memory to a region

CALLING SEQUENCE:

```
rtems_status_code rtems_region_extend(
  rtems_id            id,
  void               *starting_address,
  rtems_unsigned32    length
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - region extended successfully
`RTEMS_INVALID_ID` - invalid region id
`RTEMS_INVALID_ADDRESS` - invalid address of area to add

DESCRIPTION:

This directive adds the memory which starts at starting_address for length bytes to the region specified by id.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the region. Any local task that knows the region id can extend the region.

13.4.5 REGION_GET_SEGMENT - Get segment from a region

CALLING SEQUENCE:

```
rtems_status_code rtems_region_get_segment(
  rtems_id           id,
  rtems_unsigned32   size,
  rtems_option       option_set,
  rtems_interval     timeout,
  void             **segment
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - segment obtained successfully
`RTEMS_INVALID_ID` - invalid region id
`RTEMS_INVALID_SIZE` - request is for zero bytes or exceeds the size of maximum segment which is possible for this region
`RTEMS_UNSATISFIED` - segment of requested size not available
`RTEMS_TIMEOUT` - timed out waiting for segment
`RTEMS_OBJECT_WAS_DELETED` - semaphore deleted while waiting

DESCRIPTION:

This directive obtains a variable size segment from the region specified by id. The address of the allocated segment is returned in segment. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` components of the options parameter are used to specify whether the calling tasks wish to wait for a segment to become available or return immediately if no segment is available. For either option, if a sufficiently sized segment is available, then the segment is successfully acquired by returning immediately with the `RTEMS_SUCCESSFUL` status code.

If the calling task chooses to return immediately and a segment large enough is not available, then an error code indicating this fact is returned. If the calling task chooses to wait for the segment and a segment large enough is not available, then the calling task is placed on the region's segment wait queue and blocked. If the region was created with the `RTEMS_PRIORITY` option, then the calling task is inserted into the wait queue according to its priority. However, if the region was created with the `RTEMS_FIFO` option, then the calling task is placed at the rear of the wait queue.

The timeout parameter specifies the maximum interval that a task is willing to wait to obtain a segment. If timeout is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever.

NOTES:

The actual length of the allocated segment may be larger than the requested size because a segment size is always a multiple of the region's page size.

The following segment acquisition option constants are defined by RTEMS:

- `RTEMS_WAIT` - task will wait for semaphore (default)
- `RTEMS_NO_WAIT` - task should not wait

A clock tick is required to support the timeout functionality of this directive.

13.4.6 REGION_RETURN_SEGMENT - Return segment to a region

CALLING SEQUENCE:

```
rtems_status_code rtems_region_return_segment(
  rtems_id  id,
  void     *segment
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - segment returned successfully
`RTEMS_INVALID_ID` - invalid region id
`RTEMS_INVALID_ADDRESS` - segment address not in region

DESCRIPTION:

This directive returns the segment specified by segment to the region specified by id. The returned segment is merged with its neighbors to form the largest possible segment. The first task on the wait queue is examined to determine if its segment request can now be satisfied. If so, it is given a segment and unblocked. This process is repeated until the first task's segment request cannot be satisfied.

NOTES:

This directive will cause the calling task to be preempted if one or more local tasks are waiting for a segment and the following conditions exist:

- a waiting task has a higher priority than the calling task
- the size of the segment required by the waiting task is less than or equal to the size of the segment returned.

13.4.7 REGION_GET_SEGMENT_SIZE - Obtain size of a segment

CALLING SEQUENCE:

```
rtems_status_code rtems_region_get_segment_size(
  rtems_id           id,
  void              *segment,
  rtems_unsigned32   *size
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - segment obtained successfully
`RTEMS_INVALID_ID` - invalid region id
`RTEMS_INVALID_ADDRESS` - segment address not in region

DESCRIPTION:

This directive obtains the size in bytes of the specified segment.

NOTES:

The actual length of the allocated segment may be larger than the requested size because a segment size is always a multiple of the region's page size.

14 Dual-Ported Memory Manager

14.1 Introduction

The dual-ported memory manager provides a mechanism for converting addresses between internal and external representations for multiple dual-ported memory areas (DPMA). The directives provided by the dual-ported memory manager are:

- `rtems_port_create` - Create a port
- `rtems_port_ident` - Get ID of a port
- `rtems_port_delete` - Delete a port
- `rtems_port_external_to_internal` - Convert external to internal address
- `rtems_port_internal_to_external` - Convert internal to external address

14.2 Background

A dual-ported memory area (DPMA) is an contiguous block of RAM owned by a particular processor but which can be accessed by other processors in the system. The owner accesses the memory using internal addresses, while other processors must use external addresses. RTEMS defines a port as a particular mapping of internal and external addresses.

There are two system configurations in which dual-ported memory is commonly found. The first is tightly-coupled multiprocessor computer systems where the dual-ported memory is shared between all nodes and is used for inter-node communication. The second configuration is computer systems with intelligent peripheral controllers. These controllers typically utilize the DPMA for high-performance data transfers.

14.3 Operations

14.3.1 Creating a Port

The `rtems_port_create` directive creates a port into a DPMA with the user-defined name. The user specifies the association between internal and external representations for the port being created. RTEMS allocates a Dual-Ported Memory Control Block (DPCB) from the DPCB free list to maintain the newly created DPMA. RTEMS also generates a unique dual-ported memory port ID which is returned to the calling task. RTEMS does not initialize the dual-ported memory area or access any memory within it.

14.3.2 Obtaining Port IDs

When a port is created, RTEMS generates a unique port ID and assigns it to the created port until it is deleted. The port ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_port_create` directive, the task ID is stored in a user provided location. Second, the port ID may be obtained later using the `rtems_port_ident` directive. The port ID is used by other dual-ported memory manager directives to access this port.

14.3.3 Converting an Address

The `rtems_port_external_to_internal` directive is used to convert an address from external to internal representation for the specified port. The `rtems_port_internal_to_external` directive is used to convert an address from internal to external representation for the specified port. If an attempt is made to convert an address which lies outside the specified DPMA, then the address to be converted will be returned.

14.3.4 Deleting a DPMA Port

A port can be removed from the system and returned to RTEMS with the `rtems_port_delete` directive. When a port is deleted, its control block is returned to the DPCB free list.

14.4 Directives

This section details the dual-ported memory manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

14.4.1 PORT_CREATE - Create a port

CALLING SEQUENCE:

```
rtems_status_code rtems_port_create(
  rtems_name        name,
  void             *internal_start,
  void             *external_start,
  rtems_unsigned32  length,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - port created successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_INVALID_ADDRESS` - address not on four byte boundary
`RTEMS_TOO_MANY` - too many DP memory areas created

DESCRIPTION:

This directive creates a port which resides on the local node for the specified DPMA. The assigned port id is returned in id. This port id is used as an argument to other dual-ported memory manager directives to convert addresses within this DPMA.

For control and maintenance of the port, RTEMS allocates and initializes an DPCB from the DPCB free pool. Thus memory from the dual-ported memory area is not used to store the DPCB.

NOTES:

The internal_address and external_address parameters must be on a four byte boundary.

This directive will not cause the calling task to be preempted.

14.4.2 PORT_IDENT - Get ID of a port

CALLING SEQUENCE:

```
rtems_status_code rtems_port_ident(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - port identified successfully
`RTEMS_INVALID_NAME` - port name not found

DESCRIPTION:

This directive obtains the port id associated with the specified name to be acquired. If the port name is not unique, then the port id will match one of the DPMAs with that name. However, this port id is not guaranteed to correspond to the desired DPMA. The port id is used to access this DPMA in other dual-ported memory area related directives.

NOTES:

This directive will not cause the running task to be preempted.

14.4.3 PORT_DELETE - Delete a port

CALLING SEQUENCE:

```
rtems_status_code rtems_port_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - port deleted successfully
`RTEMS_INVALID_ID` - invalid port id

DESCRIPTION:

This directive deletes the dual-ported memory area specified by id. The DPCB for the deleted dual-ported memory area is reclaimed by RTEMS.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the port. Any local task that knows the port id can delete the port.

14.4.4 PORT_EXTERNAL_TO_INTERNAL - Convert external to internal address

CALLING SEQUENCE:

```
rtems_status_code rtems_port_external_to_internal(
  rtems_id   id,
  void      *external,
  void     **internal
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - always successful

DESCRIPTION:

This directive converts a dual-ported memory address from external to internal representation for the specified port. If the given external address is invalid for the specified port, then the internal address is set to the given external address.

NOTES:

This directive is callable from an ISR.

This directive will not cause the calling task to be preempted.

14.4.5 PORT_INTERNAL_TO_EXTERNAL - Convert internal to external address

CALLING SEQUENCE:

```
rtems_status_code rtems_port_internal_to_external(
  rtems_id   id,
  void      *internal,
  void     **external
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - always successful

DESCRIPTION:

This directive converts a dual-ported memory address from internal to external representation so that it can be passed to owner of the DPMA represented by the specified port. If the given internal address is an invalid dual-ported address, then the external address is set to the given internal address.

NOTES:

This directive is callable from an ISR.

This directive will not cause the calling task to be preempted.

15 I/O Manager

15.1 Introduction

The input/output interface manager provides a well-defined mechanism for accessing device drivers and a structured methodology for organizing device drivers. The directives provided by the I/O manager are:

- `rtems_io_initialize` - Initialize a device driver
- `rtems_io_register_name` - Register a device name
- `rtems_io_lookup_name` - Look up a device name
- `rtems_io_open` - Open a device
- `rtems_io_close` - Close a device
- `rtems_io_read` - Read from a device
- `rtems_io_write` - Write to a device
- `rtems_io_control` - Special device services

15.2 Background

15.2.1 Device Driver Table

Each application utilizing the RTEMS I/O manager must specify the address of a Device Driver Table in its Configuration Table. This table contains each device driver's entry points. Each device driver may contain the following entry points:

- Initialization
- Open
- Close
- Read
- Write
- Control

If the device driver does not support a particular entry point, then that entry in the Configuration Table should be NULL. RTEMS will return `RTEMS_SUCCESSFUL` as the executive's and zero (0) as the device driver's return code for these device driver entry points.

15.2.2 Major and Minor Device Numbers

Each call to the I/O manager must provide a device's major and minor numbers as arguments. The major number is the index of the requested driver's entry points in the Device Driver Table, and is used to select a specific device driver. The exact usage of the minor number is driver specific, but is commonly used to distinguish between a number of devices controlled by the same driver.

15.2.3 Device Names

The I/O Manager provides facilities to associate a name with a particular device. Directives are provided to register the name of a device and to look up the major/minor number pair associated with a device name.

15.2.4 Device Driver Environment

Application developers, as well as device driver developers, must be aware of the following regarding the RTEMS I/O Manager:

- A device driver routine executes in the context of the invoking task. Thus if the driver blocks, the invoking task blocks.
- The device driver is free to change the modes of the invoking task, although the driver should restore them to their original values.
- Device drivers may be invoked from ISRs.
- Only local device drivers are accessible through the I/O manager.
- A device driver routine may invoke all other RTEMS directives, including I/O directives, on both local and global objects.

Although the RTEMS I/O manager provides a framework for device drivers, it makes no assumptions regarding the construction or operation of a device driver.

15.2.5 Device Driver Interface

When an application invokes an I/O manager directive, RTEMS determines which device driver entry point must be invoked. The information passed by the application to RTEMS is then passed to the correct device driver entry point. RTEMS will invoke each device driver entry point assuming it is compatible with the following prototype:

```
rtems_device_driver io_entry(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument_block
);
```

The format and contents of the parameter block are device driver and entry point dependent.

It is recommended that a device driver avoid generating error codes which conflict with those used by application components. A common technique used to generate driver specific error codes is to make the most significant part of the status indicate a driver specific code.

15.2.6 Device Driver Initialization

RTEMS automatically initializes all device drivers when multitasking is initiated via the initialize_executive directive. RTEMS initializes the device drivers by invoking each device driver initialization entry point with the following parameters:

major	the major device number for this device driver.
minor	zero.
argument_block	will point to the Configuration Table.

The returned status will be ignored by RTEMS. If the driver cannot successfully initialize the device, then it should invoke the fatal_error_occurred directive.

15.3 Operations

15.3.1 Register and Lookup Name

The `rtems_io_register` directive associates a name with the specified device (i.e. major/minor number pair). Device names are typically registered as part of the device driver initialization sequence. The `rtems_io_lookup` directive is used to determine the major/minor number pair associated with the specified device name. The use of these directives frees the application from being dependent on the arbitrary assignment of major numbers in a particular application. No device naming conventions are dictated by RTEMS.

15.3.2 Accessing an Device Driver

The I/O manager provides directives which enable the application program to utilize device drivers in a standard manner. There is a direct correlation between the RTEMS I/O manager directives `rtems_io_initialize`, `rtems_io_open`, `rtems_io_close`, `rtems_io_read`, `rtems_io_write`, and `rtems_io_control` and the underlying device driver entry points.

15.4 Directives

This section details the I/O manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

15.4.1 IO_INITIALIZE - Initialize a device driver

CALLING SEQUENCE:

```
rtems_status_code rtems_io_initialize(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - successfully initialized
`RTEMS_INVALID_NUMBER` - invalid major device number

DESCRIPTION:

This directive calls the device driver initialization routine specified in the Device Driver Table for this major number. This directive is automatically invoked for each device driver when multitasking is initiated via the initialize_executive directive.

A device driver initialization module is responsible for initializing all hardware and data structures associated with a device. If necessary, it can allocate memory to be used during other operations.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being initialized.

15.4.2 IO_REGISTER_NAME - Register a device

CALLING SEQUENCE:

```
rtems_status_code rtems_io_register_name(
  char                      *name,
  rtems_device_major_number  major,
  rtems_device_minor_number  minor
);
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - successfully initialized
RTEMS_TOO_MANY - too many devices registered

DESCRIPTION:

This directive associates name with the specified major/minor number pair.

NOTES:

This directive will not cause the calling task to be preempted.

15.4.3 IO_LOOKUP_NAME - Lookup a device

CALLING SEQUENCE:

```
rtems_status_code rtems_io_lookup_name(
  const char                *name,
  rtems_driver_name_t      **device_info
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - successfully initialized
`RTEMS_UNSATISFIED` - name not registered

DESCRIPTION:

This directive returns the major/minor number pair associated with the given device name in device_info.

NOTES:

This directive will not cause the calling task to be preempted.

15.4.4 IO_OPEN - Open a device

CALLING SEQUENCE:

```
rtems_status_code rtems_io_open(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument
);
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - successfully initialized
RTEMS_INVALID_NUMBER - invalid major device number

DESCRIPTION:

This directive calls the device driver open routine specified in the Device Driver Table for this major number. The open entry point is commonly used by device drivers to provide exclusive access to a device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

15.4.5 IO_CLOSE - Close a device

CALLING SEQUENCE:

```
rtems_status_code rtems_io_close(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - successfully initialized
`RTEMS_INVALID_NUMBER` - invalid major device number

DESCRIPTION:

This directive calls the device driver close routine specified in the Device Driver Table for this major number. The close entry point is commonly used by device drivers to relinquish exclusive access to a device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

15.4.6 IO_READ - Read from a device

CALLING SEQUENCE:

```
rtems_status_code rtems_io_read(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - successfully initialized
`RTEMS_INVALID_NUMBER` - invalid major device number

DESCRIPTION:

This directive calls the device driver read routine specified in the Device Driver Table for this major number. Read operations typically require a buffer address as part of the argument parameter block. The contents of this buffer will be replaced with data from the device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

15.4.7 IO_WRITE - Write to a device

CALLING SEQUENCE:

```
rtems_status_code rtems_io_write(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - successfully initialized
`RTEMS_INVALID_NUMBER` - invalid major device number

DESCRIPTION:

This directive calls the device driver write routine specified in the Device Driver Table for this major number. Write operations typically require a buffer address as part of the argument parameter block. The contents of this buffer will be sent to the device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

15.4.8 IO_CONTROL - Special device services

CALLING SEQUENCE:

```
rtems_status_code rtems_io_control(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - successfully initialized
`RTEMS_INVALID_NUMBER` - invalid major device number

DESCRIPTION:

This directive calls the device driver I/O control routine specified in the Device Driver Table for this major number. The exact functionality of the driver entry called by this directive is driver dependent. It should not be assumed that the control entries of two device drivers are compatible. For example, an RS-232 driver I/O control operation may change the baud rate of a serial line, while an I/O control operation for a floppy disk driver may cause a seek operation.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

16 Fatal Error Manager

16.1 Introduction

The fatal error manager processes all fatal or irrecoverable errors. The directive provided by the fatal error manager is:

- `rtems_fatal_error_occurred` - Invoke the fatal error handler

16.2 Background

The fatal error manager is called upon detection of an irrecoverable error condition by either RTEMS or the application software. Fatal errors can be detected from three sources:

- the executive (RTEMS)
- user system code
- user application code

RTEMS automatically invokes the fatal error manager upon detection of an error it considers to be fatal. Similarly, the user should invoke the fatal error manager upon detection of a fatal error.

Each status or dynamic user extension set may include a fatal error handler. The fatal error handler in the static extension set can be used to provide access to debuggers and monitors which may be present on the target hardware. If any user-supplied fatal error handlers are installed, the fatal error manager will invoke them. If no user handlers are configured or if all the user handler return control to the fatal error manager, then the RTEMS default fatal error handler is invoked. If the default fatal error handler is invoked, then the system state is marked as failed.

Although the precise behavior of the default fatal error handler is processor specific, in general, it will disable all maskable interrupts, place the error code in a known processor dependent place (generally either on the stack or in a register), and halt the processor. The precise actions of the RTEMS fatal error are discussed in the Default Fatal Error Processing chapter of the Applications Supplement document for a specific target processor.

16.3 Operations

16.3.1 Announcing a Fatal Error

The `rtems_fatal_error_occurred` directive is invoked when a fatal error is detected. Before invoking any user-supplied fatal error handlers or the RTEMS fatal error handler, the `rtems_fatal_error_occurred` directive stores useful information in the variable `_Internal_errors_What_happened`. This structure contains three pieces of information:

- the source of the error (API or executive core),
- whether the error was generated internally by the executive, and a
- a numeric code to indicate the error type.

The error type indicator is dependent on the source of the error and whether or not the error was internally generated by the executive. If the error was generated from an API, then the error code will be of that API's error or status codes. The status codes for the RTEMS API are in c/src/exec/rtems/headers/status.h. Those for the POSIX API can be found in <errno.h>.

The `rtems_fatal_error_occurred` directive is responsible for invoking an optional user-supplied fatal error handler and/or the RTEMS fatal error handler. All fatal error handlers are passed an error code to describe the error detected.

Occasionally, an application requires more sophisticated fatal error processing such as passing control to a debugger. For these cases, a user-supplied fatal error handler can be specified in the RTEMS configuration table. The User Extension Table field fatal contains the address of the fatal error handler to be executed when the `rtems_fatal_error_occurred` directive is called. If the field is set to NULL or if the configured fatal error handler returns to the executive, then the default handler provided by RTEMS is executed. This default handler will halt execution on the processor where the error occurred.

16.4 Directives

This section details the fatal error manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

16.4.1 FATAL_ERROR_OCCURRED - Invoke the fatal error handler

CALLING SEQUENCE:

```
void volatile rtems_fatal_error_occurred(
  rtems_unsigned32        the_error
);
```

DIRECTIVE STATUS CODES

NONE

DESCRIPTION:

This directive processes fatal errors. If the FATAL error extension is defined in the configuration table, then the user-defined error extension is called. If configured and the provided FATAL error extension returns, then the RTEMS default error handler is invoked. This directive can be invoked by RTEMS or by the user's application code including initialization tasks, other tasks, and ISRs.

NOTES:

This directive supports local operations only.

Unless the user-defined error extension takes special actions such as restarting the calling task, this directive WILL NOT RETURN to the caller.

The user-defined extension for this directive may wish to initiate a global shutdown.

17 Scheduling Concepts

17.1 Introduction

The concept of scheduling in real-time systems dictates the ability to provide immediate response to specific external events, particularly the necessity of scheduling tasks to run within a specified time limit after the occurrence of an event. For example, software embedded in life-support systems used to monitor hospital patients must take instant action if a change in the patient's status is detected.

The component of RTEMS responsible for providing this capability is appropriately called the scheduler. The scheduler's sole purpose is to allocate the all important resource of processor time to the various tasks competing for attention. The RTEMS scheduler allocates the processor using a priority-based, preemptive algorithm augmented to provide round-robin characteristics within individual priority groups. The goal of this algorithm is to guarantee that the task which is executing on the processor at any point in time is the one with the highest priority among all tasks in the ready state.

There are two common methods of accomplishing the mechanics of this algorithm. Both ways involve a list or chain of tasks in the ready state. One method is to randomly place tasks in the ready chain forcing the scheduler to scan the entire chain to determine which task receives the processor. The other method is to schedule the task by placing it in the proper place on the ready chain based on the designated scheduling criteria at the time it enters the ready state. Thus, when the processor is free, the first task on the ready chain is allocated the processor. RTEMS schedules tasks using the second method to guarantee faster response times to external events.

17.2 Scheduling Mechanisms

RTEMS provides four mechanisms which allow the user to impact the task scheduling process:

- user-selectable task priority level
- task preemption control
- task timeslicing control
- manual round-robin selection

Each of these methods provides a powerful capability to customize sets of tasks to satisfy the unique and particular requirements encountered in custom real-time applications. Although each mechanism operates independently, there is a precedence relationship which governs the effects of scheduling modifications. The evaluation order for scheduling characteristics is always priority, preemption mode, and timeslicing. When reading the descriptions of timeslicing and manual round-robin it is important to keep in mind that preemption (if enabled) of a task by higher priority tasks will occur as required, overriding the other factors presented in the description.

17.2.1 Task Priority and Scheduling

The most significant of these mechanisms is the ability for the user to assign a priority level to each individual task when it is created and to alter a task's priority at run-time. RTEMS provides 255 priority levels. Level 255 is the lowest priority and level 1 is the highest. When a task is added to the ready chain, it is placed behind all other tasks of the same priority. This rule provides a round-robin within priority group scheduling characteristic. This means that in a group of equal priority tasks, tasks will execute in the order they become ready or FIFO order. Even though there are ways to manipulate and adjust task priorities, the most important rule to remember is:

> **The RTEMS scheduler will always select the highest priority task that is ready to run when allocating the processor to a task.**

17.2.2 Preemption

Another way the user can alter the basic scheduling algorithm is by manipulating the preemption mode flag (`RTEMS_PREEMPT_MASK`) of individual tasks. If preemption is disabled for a task (`RTEMS_NO_PREEMPT`), then the task will not relinquish control of the processor until it terminates, blocks, or re-enables preemption. Even tasks which become ready to run and possess higher priority levels will not be allowed to execute. Note that the preemption setting has no effect on the manner in which a task is scheduled. It only applies once a task has control of the processor.

17.2.3 Timeslicing

Timeslicing or round-robin scheduling is an additional method which can be used to alter the basic scheduling algorithm. Like preemption, timeslicing is specified on a task by task basis using the timeslicing mode flag (`RTEMS_TIMESLICE_MASK`). If timeslicing is enabled for a task (`RTEMS_TIMESLICE`), then RTEMS will limit the amount of time the task can execute before the processor is allocated to another task. Each tick of the real-time clock reduces the currently running task's timeslice. When the execution time equals the timeslice, RTEMS will dispatch another task of the same priority to execute. If there are no other tasks of the same priority ready to execute, then the current task is allocated an additional timeslice and continues to run. Remember that a higher priority task will preempt the task (unless preemption is disabled) as soon as it is ready to run, even if the task has not used up its entire timeslice.

17.2.4 Manual Round-Robin

The final mechanism for altering the RTEMS scheduling algorithm is called manual round-robin. Manual round-robin is invoked by using the `rtems_task_wake_after` directive with a time interval of `RTEMS_YIELD_PROCESSOR`. This allows a task to give up the processor and be immediately returned to the ready chain at the end of its priority group. If no other tasks of the same priority are ready to run, then the task does not lose control of the processor.

17.2.5 Dispatching Tasks

The dispatcher is the RTEMS component responsible for allocating the processor to a ready task. In order to allocate the processor to one task, it must be deallocated or retrieved from the task currently using it. This involves a concept called a context switch. To perform a context switch, the dispatcher saves the context of the current task and restores the context of the task which has been allocated to the processor. Saving and restoring a task's context is the storing/loading of all the essential information about a task to enable it to continue execution without any effects of the interruption. For example, the contents of a task's register set must be the same when it is given the processor as they were when it was taken away. All of the information that must be saved or restored for a context switch is located either in the TCB or on the task's stacks.

Tasks that utilize a numeric coprocessor and are created with the `RTEMS_FLOATING_POINT` attribute require additional operations during a context switch. These additional operations are necessary to save and restore the floating point context of `RTEMS_FLOATING_POINT` tasks. To avoid unnecessary save and restore operations, the state of the numeric coprocessor is only saved when a `RTEMS_FLOATING_POINT` task is dispatched and that task was not the last task to utilize the coprocessor.

17.3 Task State Transitions

Tasks in an RTEMS system must always be in one of the five allowable task states. These states are: executing, ready, blocked, dormant, and non-existent.

```
+--------------------------------------------------------------+
|                         Non-existent                         |
|  +--------------------------------------------------------+  |
|  |                                                        |  |
|  |                                                        |  |
|  |      Creating        +---------+     Deleting          |  |
|  | -------------------> | Dormant | --------------------> |  |
|  |                      +---------+                       |  |
|  |                           |                            |  |
|  |                  Starting |                            |  |
|  |                           |                            |  |
|  |                           V          Deleting          |  |
|  |             +-------> +-------+ -------------------->  |  |
|  |  Yielding  /   +----- | Ready | ------+                |  |
|  |           /   /       +-------+ <--+   \               |  |
|  |          /   /                     \   \ Blocking      |  |
|  |         /   / Dispatching   Readying \   \             |  |
|  |        /   V                          \   V            |  |
|  |      +-----------+    Blocking     +---------+         |  |
|  |      | Executing | --------------> | Blocked |         |  |
|  |      +-----------+                 +---------+         |  |
|  |                                                        |  |
|  |                                                        |  |
|  +--------------------------------------------------------+  |
|                         Non-existent                         |
+--------------------------------------------------------------+
```

A task occupies the non-existent state before a `rtems_task_create` has been issued on its behalf. A task enters the non-existent state from any other state in the system when it is deleted with the `rtems_task_delete` directive. While a task occupies this state it does not have a TCB or a task ID assigned to it; therefore, no other tasks in the system may reference this task.

When a task is created via the `rtems_task_create` directive it enters the dormant state. This state is not entered through any other means. Although the task exists in the system, it cannot actively compete for system resources. It will remain in the dormant state until it is started via the `rtems_task_start` directive, at which time it enters the ready state. The task is now permitted to be scheduled for the processor and to compete for other system resources.

A task occupies the blocked state whenever it is unable to be scheduled to run. A running task may block itself or be blocked by other tasks in the system. The running task blocks itself through voluntary operations that cause the task to wait. The only way a task can block a task other than itself is with the `rtems_task_suspend` directive. A task enters the blocked state due to any of the following conditions:

- A task issues a `rtems_task_suspend` directive which blocks either itself or another task in the system.
- The running task issues a `rtems_message_queue_receive` directive with the wait option

and the message queue is empty.

- The running task issues an `rtems_event_receive` directive with the wait option and the currently pending events do not satisfy the request.
- The running task issues a `rtems_semaphore_obtain` directive with the wait option and the requested semaphore is unavailable.
- The running task issues a `rtems_task_wake_after` directive which blocks the task for the given time interval. If the time interval specified is zero, the task yields the processor and remains in the ready state.
- The running task issues a `rtems_task_wake_when` directive which blocks the task until the requested date and time arrives.
- The running task issues a `rtems_region_get_segment` directive with the wait option and there is not an available segment large enough to satisfy the task's request.
- The running task issues a `rtems_rate_monotonic_period` directive and must wait for the specified rate monotonic period to conclude.

A blocked task may also be suspended. Therefore, both the suspension and the blocking condition must be removed before the task becomes ready to run again.

A task occupies the ready state when it is able to be scheduled to run, but currently does not have control of the processor. Tasks of the same or higher priority will yield the processor by either becoming blocked, completing their timeslice, or being deleted. All tasks with the same priority will execute in FIFO order. A task enters the ready state due to any of the following conditions:

- A running task issues a `rtems_task_resume` directive for a task that is suspended and the task is not blocked waiting on any resource.
- A running task issues a `rtems_message_queue_send`, `rtems_message_queue_broadcast`, or a `rtems_message_queue_urgent` directive which posts a message to the queue on which the blocked task is waiting.
- A running task issues an `rtems_event_send` directive which sends an event condition to a task which is blocked waiting on that event condition.
- A running task issues a `rtems_semaphore_release` directive which releases the semaphore on which the blocked task is waiting.
- A timeout interval expires for a task which was blocked by a call to the `rtems_task_wake_after` directive.
- A timeout period expires for a task which blocked by a call to the `rtems_task_wake_when` directive.
- A running task issues a `rtems_region_return_segment` directive which releases a segment to the region on which the blocked task is waiting and a resulting segment is large enough to satisfy the task's request.

- A rate monotonic period expires for a task which blocked by a call to the `rtems_rate_monotonic_period` directive.
- A timeout interval expires for a task which was blocked waiting on a message, event, semaphore, or segment with a timeout specified.
- A running task issues a directive which deletes a message queue, a semaphore, or a region on which the blocked task is waiting.
- A running task issues a `rtems_task_restart` directive for the blocked task.
- The running task, with its preemption mode enabled, may be made ready by issuing any of the directives that may unblock a task with a higher priority. This directive may be issued from the running task itself or from an ISR.

  A ready task occupies the executing state when it has control of the CPU. A task enters the executing state due to any of the following conditions:
- The task is the highest priority ready task in the system.
- The running task blocks and the task is next in the scheduling queue. The task may be of equal priority as in round-robin scheduling or the task may possess the highest priority of the remaining ready tasks.
- The running task may reenable its preemption mode and a task exists in the ready queue that has a higher priority than the running task.
- The running task lowers its own priority and another task is of higher priority as a result.
- The running task raises the priority of a task above its own and the running task is in preemption mode.

18 Rate Monotonic Manager

18.1 Introduction

The rate monotonic manager provides facilities to implement tasks which execute in a periodic fashion. The directives provided by the rate monotonic manager are:

- `rtems_rate_monotonic_create` - Create a rate monotonic period
- `rtems_rate_monotonic_ident` - Get ID of a period
- `rtems_rate_monotonic_cancel` - Cancel a period
- `rtems_rate_monotonic_delete` - Delete a rate monotonic period
- `rtems_rate_monotonic_period` - Conclude current/Start next period
- `rtems_rate_monotonic_get_status` - Obtain status information on period

18.2 Background

The rate monotonic manager provides facilities to manage the execution of periodic tasks. This manager was designed to support application designers who utilize the Rate Monotonic Scheduling Algorithm (RMS) to insure that their periodic tasks will meet their deadlines, even under transient overload conditions. Although designed for hard real-time systems, the services provided by the rate monotonic manager may be used by any application which requires periodic tasks.

18.2.1 Rate Monotonic Manager Required Support

A clock tick is required to support the functionality provided by this manager.

18.2.2 Rate Monotonic Manager Definitions

A periodic task is one which must be executed at a regular interval. The interval between successive iterations of the task is referred to as its period. Periodic tasks can be characterized by the length of their period and execution time. The period and execution time of a task can be used to determine the processor utilization for that task. Processor utilization is the percentage of processor time used and can be calculated on a per-task or system-wide basis. Typically, the task's worst-case execution time will be less than its period. For example, a periodic task's requirements may state that it should execute for 10 milliseconds every 100 milliseconds. Although the execution time may be the average, worst, or best case, the worst-case execution time is more appropriate for use when analyzing system behavior under transient overload conditions.

In contrast, an aperiodic task executes at irregular intervals and has only a soft deadline. In other words, the deadlines for aperiodic tasks are not rigid, but adequate response times are desirable. For example, an aperiodic task may process user input from a terminal.

Finally, a sporadic task is an aperiodic task with a hard deadline and minimum interarrival time. The minimum interarrival time is the minimum period of time which exists between successive iterations of the task. For example, a sporadic task could be used to process the pressing of a fire button on a joystick. The mechanical action of the fire button insures a minimum time period between successive activations, but the missile must be launched by a hard deadline.

18.2.3 Rate Monotonic Scheduling Algorithm

The Rate Monotonic Scheduling Algorithm (RMS) is important to real-time systems designers because it allows one to guarantee that a set of tasks is schedulable. A set of tasks is said to be schedulable if all of the tasks can meet their deadlines. RMS provides a set of rules which can be used to perform a guaranteed schedulability analysis for a task set. This analysis determines whether a task set is schedulable under worst-case conditions and emphasizes the predictability of the system's behavior. It has been proven that:

> **RMS is an optimal static priority algorithm for scheduling independent, preemptible, periodic tasks on a single processor.**

RMS is optimal in the sense that if a set of tasks can be scheduled by any static priority algorithm, then RMS will be able to schedule that task set. RMS bases it schedulability analysis on the processor utilization level below which all deadlines can be met.

RMS calls for the static assignment of task priorities based upon their period. The shorter a task's period, the higher its priority. For example, a task with a 1 millisecond period has higher priority than a task with a 100 millisecond period. If two tasks have the same period, then RMS does not distinguish between the tasks. However, RTEMS specifies that when given tasks of equal priority, the task which has been ready longest will execute first. RMS's priority assignment scheme does not provide one with exact numeric values for task priorities. For example, consider the following task set and priority assignments:

Task	**Period (in milliseconds)**	**Priority**
1	100	Low
2	50	Medium
3	50	Medium
4	25	High

RMS only calls for task 1 to have the lowest priority, task 4 to have the highest priority, and tasks 2 and 3 to have an equal priority between that of tasks 1 and 4. The actual RTEMS priorities assigned to the tasks must only adhere to those guidelines.

Many applications have tasks with both hard and soft deadlines. The tasks with hard deadlines are typically referred to as the critical task set, with the soft deadline tasks being the non-critical task set. The critical task set can be scheduled using RMS, with the non-critical tasks not executing under transient overload, by simply assigning priorities such that the lowest priority critical task (i.e. longest period) has a higher priority than the highest priority non-critical task. Although RMS may be used to assign priorities to the non-critical tasks, it is not necessary. In this instance, schedulability is only guaranteed for the critical task set.

18.2.4 Schedulability Analysis

RMS allows application designers to insure that tasks can meet all deadlines, even under transient overload, without knowing exactly when any given task will execute by applying proven schedulability analysis rules.

18.2.4.1 Assumptions

The schedulability analysis rules for RMS were developed based on the following assumptions:

- The requests for all tasks for which hard deadlines exist are periodic, with a constant interval between requests.
- Each task must complete before the next request for it occurs.
- The tasks are independent in that a task does not depend on the initiation or completion of requests for other tasks.
- The execution time for each task without preemption or interruption is constant and does not vary.
- Any non-periodic tasks in the system are special. These tasks displace periodic tasks while executing and do not have hard, critical deadlines.

Once the basic schedulability analysis is understood, some of the above assumptions can be relaxed and the side-effects accounted for.

18.2.4.2 Processor Utilization Rule

The Processor Utilization Rule requires that processor utilization be calculated based upon the period and execution time of each task. The fraction of processor time spent executing task index is Time(index) / Period(index). The processor utilization can be calculated as follows:

```
Utilization = 0

for index = 1 to maximum_tasks
  Utilization = Utilization + (Time(index)/Period(index))
```

To insure schedulability even under transient overload, the processor utilization must adhere to the following rule:

```
Utilization = maximum_tasks * (2(1/maximum_tasks) - 1)
```

As the number of tasks increases, the above formula approaches ln(2) for a worst-case utilization factor of approximately 0.693. Many tasks sets can be scheduled with a greater utilization factor. In fact, the average processor utilization threshold for a randomly generated task set is approximately 0.88.

18.2.4.3 Processor Utilization Rule Example

This example illustrates the application of the Processor Utilization Rule to an application with three critical periodic tasks. The following table details the RMS priority, period, execution time, and processor utilization for each task:

Task	RMS Priority	Period	Execution Time	Processor Utilization
1	High	100	15	0.15
2	Medium	200	50	0.25
3	Low	300	100	0.33

The total processor utilization for this task set is 0.73 which is below the upper bound of 3 * (2(1/3) - 1), or 0.779, imposed by the Processor Utilization Rule. Therefore, this task set is guaranteed to be schedulable using RMS.

18.2.4.4 First Deadline Rule

If a given set of tasks do exceed the processor utilization upper limit imposed by the Processor Utilization Rule, they can still be guaranteed to meet all their deadlines by application of the First Deadline Rule. This rule can be stated as follows:

For a given set of independent periodic tasks, if each task meets its first deadline when all tasks are started at the same time, then the deadlines will always be met for any combination of start times.

A key point with this rule is that ALL periodic tasks are assumed to start at the exact same instant in time. Although this assumption may seem to be invalid, RTEMS makes it quite easy to insure. By having a non-preemptible user initialization task, all application tasks, regardless of priority, can be created and started before the initialization deletes itself. This technique insures that all tasks begin to compete for execution time at the same instant – when the user initialization task deletes itself.

18.2.4.5 First Deadline Rule Example

The First Deadline Rule can insure schedulability even when the Processor Utilization Rule fails. The example below is a modification of the Processor Utilization Rule example where task execution time has been increased from 15 to 25 units. The following table details the RMS priority, period, execution time, and processor utilization for each task:

Task	**RMS Priority**	**Period**	**Execution Time**	**Processor Utilization**
1	High	100	25	0.25
2	Medium	200	50	0.25
3	Low	300	100	0.33

The total processor utilization for the modified task set is 0.83 which is above the upper bound of 3 * (2(1/3) - 1), or 0.779, imposed by the Processor Utilization Rule. Therefore, this task set is not guaranteed to be schedulable using RMS. However, the First Deadline Rule can guarantee the schedulability of this task set. This rule calls for one to examine each occurrence of deadline until either all tasks have met their deadline or one task failed to meet its first deadline. The following table details the time of each deadline occurrence, the maximum number of times each task may have run, the total execution time, and whether all the deadlines have been met.

Deadline Time	**Task 1**	**Task 2**	**Task 3**	**Total Execution Time**	**All Deadlines Net?**
100	1	1	1	25 + 50 + 100 = 175	NO
200	2	1	1	50 + 50 + 100 = 200	YES

The key to this analysis is to recognize when each task will execute. For example at time 100, task 1 must have met its first deadline, but tasks 2 and 3 may also have begun execution. In this example, at time 100 tasks 1 and 2 have completed execution and thus have met their first deadline. Tasks 1 and 2 have used (25 + 50) = 75 time units, leaving (100 - 75) = 25 time units for task 3 to begin. Because task 3 takes 100 ticks to execute, it will not have completed execution at time 100. Thus at time 100, all of the tasks except task 3 have met their first deadline.

At time 200, task 1 must have met its second deadline and task 2 its first deadline. As a result, of the first 200 time units, task 1 uses (2 * 25) = 50 and task 2 uses 50, leaving (200 - 100) time units for task 3. Task 3 requires 100 time units to execute, thus it will have completed execution at time 200. Thus, all of the tasks have met their first deadlines at time 200, and the task set is schedulable using the First Deadline Rule.

18.2.4.6 Relaxation of Assumptions

The assumptions used to develop the RMS schedulability rules are uncommon in most real-time systems. For example, it was assumed that tasks have constant unvarying execution time. It is possible to relax this assumption, simply by using the worst-case execution time of each task.

Another assumption is that the tasks are independent. This means that the tasks do not wait for one another or contend for resources. This assumption can be relaxed by accounting for the amount of time a task spends waiting to acquire resources. Similarly, each task's execution time must account for any I/O performed and any RTEMS directive calls.

In addition, the assumptions did not account for the time spent executing interrupt service routines. This can be accounted for by including all the processor utilization by interrupt service routines in the utilization calculation. Similarly, one should also account for the impact of delays in accessing local memory caused by direct memory access and other processors accessing local dual-ported memory.

The assumption that nonperiodic tasks are used only for initialization or failure-recovery can be relaxed by placing all periodic tasks in the critical task set. This task set can be scheduled and analyzed using RMS. All nonperiodic tasks are placed in the non-critical task set. Although the critical task set can be guaranteed to execute even under transient overload, the non-critical task set is not guaranteed to execute.

In conclusion, the application designer must be fully cognizant of the system and its run-time behavior when performing schedulability analysis for a system using RMS. Every hardware and software factor which impacts the execution time of each task must be accounted for in the schedulability analysis.

18.2.4.7 Further Reading

For more information on Rate Monotonic Scheduling and its schedulability analysis, the reader is referred to the following:

> *C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment."* **Journal of the Association of Computing Machinery**. *January 1973. pp. 46-61.*
>
> *John Lehoczky, Lui Sha, and Ye Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior."* **IEEE Real-Time Systems Symposium**. *1989. pp. 166-171.*
>
> *Lui Sha and John Goodenough. "Real-Time Scheduling Theory and Ada."* **IEEE Computer**. *April 1990. pp. 53-62.*
>
> *Alan Burns. "Scheduling hard real-time systems: a review."* **Software Engineering Journal**. *May 1991. pp. 116-128.*

18.3 Operations

18.3.1 Creating a Rate Monotonic Period

The `rtems_rate_monotonic_create` directive creates a rate monotonic period which is to be used by the calling task to delineate a period. RTEMS allocates a Period Control Block (PCB) from the PCB free list. This data structure is used by RTEMS to manage the newly created rate monotonic period. RTEMS returns a unique period ID to the application which is used by other rate monotonic manager directives to access this rate monotonic period.

18.3.2 Manipulating a Period

The `rtems_rate_monotonic_period` directive is used to establish and maintain periodic execution utilizing a previously created rate monotonic period. Once initiated by the `rtems_rate_monotonic_period` directive, the period is said to run until it either expires or is reinitiated. The state of the rate monotonic period results in one of the following scenarios:

- If the rate monotonic period is running, the calling task will be blocked for the remainder of the outstanding period and, upon completion of that period, the period will be reinitiated with the specified period.
- If the rate monotonic period is not currently running and has not expired, it is initiated with a length of period ticks and the calling task returns immediately.
- If the rate monotonic period has expired before the task invokes the `rtems_rate_monotonic_period` directive, the period will be initiated with a length of period ticks and the calling task returns immediately with a timeout error status.

18.3.3 Obtaining a Period's Status

If the `rtems_rate_monotonic_period` directive is invoked with a period of `RTEMS_PERIOD_STATUS` ticks, the current state of the specified rate monotonic period will be returned. The following table details the relationship between the period's status and the directive status code returned by the `rtems_rate_monotonic_period` directive:

- `RTEMS_SUCCESSFUL` - period is running
- `RTEMS_TIMEOUT` - period has expired
- `RTEMS_NOT_DEFINED` - period has never been initiated

Obtaining the status of a rate monotonic period does not alter the state or length of that period.

18.3.4 Canceling a Period

The `rtems_rate_monotonic_cancel` directive is used to stop the period maintained by the specified rate monotonic period. The period is stopped and the rate monotonic period can be reinitiated using the `rtems_rate_monotonic_period` directive.

18.3.5 Deleting a Rate Monotonic Period

The `rtems_rate_monotonic_delete` directive is used to delete a rate monotonic period. If the period is running and has not expired, the period is automatically canceled. The rate monotonic period's control block is returned to the PCB free list when it is deleted. A rate monotonic period can be deleted by a task other than the task which created the period.

18.3.6 Examples

The following sections illustrate common uses of rate monotonic periods to construct periodic tasks.

18.3.7 Simple Periodic Task

This example consists of a single periodic task which, after initialization, executes every 100 clock ticks.

```
rtems_task Periodic_task()
{
  rtems_name        name;
  rtems_id          period;
  rtems_status_code status;

  name = rtems_build_name( 'P', 'E', 'R', 'D' );

  (void) rate_monotonic_create( name, &period );

  while ( 1 ) {
    if ( rate_monotonic_period( period, 100 ) == TIMEOUT )
      break;

    /* Perform some periodic actions */
  }

  /* missed period so delete period and SELF */

  (void) rate_monotonic_delete( period );
  (void) task_delete( SELF );
}
```

The above task creates a rate monotonic period as part of its initialization. The first time the loop is executed, the `rtems_rate_monotonic_period` directive will initiate the period for 100 ticks and return immediately. Subsequent invocations of the `rtems_rate_monotonic_period` directive will result in the task blocking for the remainder of the 100 tick period. If, for any reason, the body of the loop takes more than 100 ticks to execute, the `rtems_rate_monotonic_period` directive will return the `RTEMS_TIMEOUT` status. If the above task misses its deadline, it will delete the rate monotonic period and itself.

18.3.8 Task with Multiple Periods

This example consists of a single periodic task which, after initialization, performs two sets of actions every 100 clock ticks. The first set of actions is performed in the first forty clock ticks of every 100 clock ticks, while the second set of actions is performed between the fortieth and seventieth clock ticks. The last thirty clock ticks are not used by this task.

```
task Periodic_task()
{
  rtems_name        name_1, name_2;
  rtems_id          period_1, period_2;
  rtems_status_code status;

  name_1 = rtems_build_name( 'P', 'E', 'R', '1' );
  name_2 = rtems_build_name( 'P', 'E', 'R', '2' );

  (void ) rate_monotonic_create( name_1, &period_1 );
  (void ) rate_monotonic_create( name_2, &period_2 );

  while ( 1 ) {
    if ( rate_monotonic_period( period_1, 100 ) == TIMEOUT )
      break;

    if ( rate_monotonic_period( period_2, 40 ) == TIMEOUT )
      break;

    /*
     *  Perform first set of actions between clock
     *  ticks 0 and 39 of every 100 ticks.
     */

    if ( rate_monotonic_period( period_2, 30 ) == TIMEOUT )
      break;

    /*
     *  Perform second set of actions between clock 40 and 69
     *  of every 100 ticks.  THEN ...
     *
     *  Check to make sure we didn't miss the period_2 period.
     */

    if ( rate_monotonic_period( period_2, STATUS ) == TIMEOUT )
      break;

    (void) rate_monotonic_cancel( period_2 );
  }

  /* missed period so delete period and SELF */

  (void ) rate_monotonic_delete( period_1 );
  (void ) rate_monotonic_delete( period_2 );
  (void ) task_delete( SELF );
}
```

The above task creates two rate monotonic periods as part of its initialization. The first time the loop is executed, the `rtems_rate_monotonic_period` directive will initiate the period_1 period for 100 ticks and return immediately. Subsequent invocations of the `rtems_rate_monotonic_period` directive for period_1 will result in the task blocking for the remainder of the 100 tick period. The period_2 period is used to control the execution time of the two sets of actions within each 100 tick period established by period_1. The `rtems_rate_monotonic_cancel( period_2 )` call is performed to insure that the period_2 period does not expire while the task is blocked on the period_1 period. If this cancel operation were not performed, every time the `rtems_rate_monotonic_period( period_1, 40 )` call is executed, except for the initial one, a directive status of `RTEMS_TIMEOUT` is returned. It is important to note that every time this call is made, the period_1 period will be initiated immediately and the task will not block.

If, for any reason, the task misses any deadline, the `rtems_rate_monotonic_period` directive will return the `RTEMS_TIMEOUT` directive status. If the above task misses its deadline, it will delete the rate monotonic periods and itself.

18.4 Directives

This section details the rate monotonic manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

18.4.1 RATE_MONOTONIC_CREATE - Create a rate monotonic period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_create(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - rate monotonic period created successfully
`RTEMS_INVALID_NAME` - invalid task name
`RTEMS_TOO_MANY` - too many periods created

DESCRIPTION:

This directive creates a rate monotonic period. The assigned rate monotonic id is returned in id. This id is used to access the period with other rate monotonic manager directives. For control and maintenance of the rate monotonic period, RTEMS allocates a PCB from the local PCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

18.4.2 RATE_MONOTONIC_IDENT - Get ID of a period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_ident(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - period identified successfully
`RTEMS_INVALID_NAME` - period name not found

DESCRIPTION:

This directive obtains the period id associated with the period name to be acquired. If the period name is not unique, then the period id will match one of the periods with that name. However, this period id is not guaranteed to correspond to the desired period. The period id is used to access this period in other rate monotonic manager directives.

NOTES:

This directive will not cause the running task to be preempted.

18.4.3 RATE_MONOTONIC_CANCEL - Cancel a period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_cancel(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - period canceled successfully
`RTEMS_INVALID_ID` - invalid rate monotonic period id
`RTEMS_NOT_OWNER_OF_RESOURCE` - rate monotonic period not created by calling task

DESCRIPTION:

This directive cancels the rate monotonic period id. This period will be reinitiated by the next invocation of `rtems_rate_monotonic_period` with id.

NOTES:

This directive will not cause the running task to be preempted.

The rate monotonic period specified by id must have been created by the calling task.

18.4.4 RATE_MONOTONIC_DELETE - Delete a rate monotonic period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - period deleted successfully
`RTEMS_INVALID_ID` - invalid rate monotonic period id

DESCRIPTION:

This directive deletes the rate monotonic period specified by id. If the period is running, it is automatically canceled. The PCB for the deleted period is reclaimed by RTEMS.

NOTES:

This directive will not cause the running task to be preempted.

A rate monotonic period can be deleted by a task other than the task which created the period.

18.4.5 RATE_MONOTONIC_PERIOD - Conclude current/Start next period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_period(
  rtems_id       id,
  rtems_interval length
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - period initiated successfully
`RTEMS_INVALID_ID` - invalid rate monotonic period id
`RTEMS_NOT_OWNER_OF_RESOURCE` - period not created by calling task
`RTEMS_NOT_DEFINED` - period has never been initiated (only possible when period is set to PERIOD_STATUS)
`RTEMS_TIMEOUT` - period has expired

DESCRIPTION:

This directive initiates the rate monotonic period id with a length of period ticks. If id is running, then the calling task will block for the remainder of the period before reinitiating the period with the specified period. If id was not running (either expired or never initiated), the period is immediately initiated and the directive returns immediately.

If invoked with a period of `RTEMS_PERIOD_STATUS` ticks, the current state of id will be returned. The directive status indicates the current state of the period. This does not alter the state or period of the period.

NOTES:

This directive will not cause the running task to be preempted.

18.4.6 RATE_MONOTONIC_GET_STATUS - Obtain status information on period

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_get_status(
  rtems_id                              id,
  rtems_rate_monotonic_period_status *status
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - period initiated successfully
`RTEMS_INVALID_ID` - invalid rate monotonic period id
`RTEMS_INVALID_ADDRESS` - invalid address of status

DESCRIPTION:

This directive returns status information associated with the rate monotonic period id in the following data structure:

```
typedef struct {
  rtems_rate_monotonic_period_states  state;
  unsigned32                          ticks_since_last_period;
  unsigned32                          ticks_executed_since_last_period;
}  rtems_rate_monotonic_period_status;
```

If the period's state is `RATE_MONOTONIC_INACTIVE`, both ticks_since_last_period and ticks_executed_since_last_period will be set to 0. Otherwise, ticks_since_last_period will contain the number of clock ticks which have occurred since the last invocation of the `rtems_rate_monotonic_period` directive. Also in this case, the ticks_executed_since_last_period will indicate how much processor time the owning task has consumed since the invocation of the `rtems_rate_monotonic_period` directive.

NOTES:

This directive will not cause the running task to be preempted.

19 Board Support Packages

19.1 Introduction

A board support package (BSP) is a collection of user-provided facilities which interface RTEMS and an application with a specific hardware platform. These facilities may include hardware initialization, device drivers, user extensions, and a Multiprocessor Communications Interface (MPCI). However, a minimal BSP need only support processor reset and initialization and, if needed, a clock tick.

19.2 Reset and Initialization

An RTEMS based application is initiated or re-initiated when the processor is reset. This initialization code is responsible for preparing the target platform for the RTEMS application. Although the exact actions performed by the initialization code are highly processor and target dependent, the logical functionality of these actions are similar across a variety of processors and target platforms.

Normally, the application's initialization is performed at two separate times: before the call to `rtems_initialize_executive` (reset application initialization) and after `rtems_initialize_executive` in the user's initialization tasks (local and global application initialization). The order of the startup procedure is as follows:

1. Reset application initialization.
2. Call to `rtems_initialize_executive`
3. Local and global application initialization.

The reset application initialization code is executed first when the processor is reset. All of the hardware must be initialized to a quiescent state by this software before initializing RTEMS. When in quiescent state, devices do not generate any interrupts or require any servicing by the application. Some of the hardware components may be initialized in this code as well as any application initialization that does not involve calls to RTEMS directives.

The processor's Interrupt Vector Table which will be used by the application may need to be set to the required value by the reset application initialization code. Because interrupts are enabled automatically by RTEMS as part of the `rtems_initialize_executive` directive, the Interrupt Vector Table MUST be set before this directive is invoked to insure correct interrupt vectoring. The processor's Interrupt Vector Table must be accessible by RTEMS as it will be modified by the `rtems_interrupt_catch` directive. On some CPUs, RTEMS installs it's own Interrupt Vector Table as part of initialization and thus these requirements are met automatically. The reset code which is executed before the call to `rtems_initialize_executive` has the following requirements:

- Must not make any RTEMS directive calls.

- If the processor supports multiple privilege levels, must leave the processor in the most privileged, or supervisory, state.
- Must allocate a stack of at least `RTEMS_MINIMUM_STACK_SIZE` bytes and initialize the stack pointer for the `rtems_initialize_executive` directive.
- Must initialize the processor's Interrupt Vector Table.
- Must disable all maskable interrupts.
- If the processor supports a separate interrupt stack, must allocate the interrupt stack and initialize the interrupt stack pointer.

The `rtems_initialize_executive` directive does not return to the initialization code, but causes the highest priority initialization task to begin execution. Initialization tasks are used to perform both local and global application initialization which is dependent on RTEMS facilities. The user initialization task facility is typically used to create the application's set of tasks.

19.2.1 Interrupt Stack Requirements

The worst-case stack usage by interrupt service routines must be taken into account when designing an application. If the processor supports interrupt nesting, the stack usage must include the deepest nest level. The worst-case stack usage must account for the following requirements:

- Processor's interrupt stack frame
- Processor's subroutine call stack frame
- RTEMS system calls
- Registers saved on stack
- Application subroutine calls

The size of the interrupt stack must be greater than or equal to the constant `RTEMS_MINIMUM_STACK_SIZE`.

19.2.2 Processors with a Separate Interrupt Stack

Some processors support a separate stack for interrupts. When an interrupt is vectored and the interrupt is not nested, the processor will automatically switch from the current stack to the interrupt stack. The size of this stack is based solely on the worst-case stack usage by interrupt service routines.

The dedicated interrupt stack for the entire application is supplied and initialized by the reset and initialization code of the user's board support package. Since all ISRs use this stack, the stack size must take into account the worst case stack usage by any combination of nested ISRs.

19.2.3 Processors without a Separate Interrupt Stack

Some processors do not support a separate stack for interrupts. In this case, without special assistance every task's stack must include enough space to handle the task's worst-case stack usage as well as the worst-case interrupt stack usage. This is necessary because the worst-case interrupt nesting could occur while any task is executing.

On many processors without dedicated hardware managed interrupt stacks, RTEMS manages a dedicated interrupt stack in software. If this capability is supported on a CPU, then it is logically equivalent to the processor supporting a separate interrupt stack in hardware.

19.3 Device Drivers

Device drivers consist of control software for special peripheral devices and provide a logical interface for the application developer. The RTEMS I/O manager provides directives which allow applications to access these device drivers in a consistent fashion. A Board Support Package may include device drivers to access the hardware on the target platform. These devices typically include serial and parallel ports, counter/timer peripherals, real-time clocks, disk interfaces, and network controllers.

For more information on device drivers, refer to the I/O Manager chapter.

19.3.1 Clock Tick Device Driver

Most RTEMS applications will include a clock tick device driver which invokes the `rtems_clock_tick` directive at regular intervals. The clock tick is necessary if the application is to utilize timeslicing, the clock manager, the timer manager, the rate monotonic manager, or the timeout option on blocking directives.

The clock tick is usually provided as an interrupt from a counter/timer or a real-time clock device. When a counter/timer is used to provide the clock tick, the device is typically programmed to operate in continuous mode. This mode selection causes the device to automatically reload the initial count and continue the countdown without programmer intervention. This reduces the overhead required to manipulate the counter/timer in the clock tick ISR and increases the accuracy of tick occurrences. The initial count can be based on the microseconds_per_tick field in the RTEMS Configuration Table. An alternate approach is to set the initial count for a fixed time period (such as one millisecond) and have the ISR invoke `rtems_clock_tick` on the microseconds_per_tick boundaries. Obviously, this can induce some error if the configured microseconds_per_tick is not evenly divisible by the chosen clock interrupt quantum.

It is important to note that the interval between clock ticks directly impacts the granularity of RTEMS timing operations. In addition, the frequency of clock ticks is an important factor in the overall level of system overhead. A high clock tick frequency results in less processor time being available for task execution due to the increased number of clock tick ISRs.

19.4 User Extensions

RTEMS allows the application developer to augment selected features by invoking user-supplied extension routines when the following system events occur:

- Task creation
- Task initiation
- Task reinitiation
- Task deletion
- Task context switch
- Post task context switch
- Task begin
- Task exits
- Fatal error detection

User extensions can be used to implement a wide variety of functions including execution profiling, non-standard coprocessor support, debug support, and error detection and recovery. For example, the context of a non-standard numeric coprocessor may be maintained via the user extensions. In this example, the task creation and deletion extensions are responsible for allocating and deallocating the context area, the task initiation and reinitiation extensions would be responsible for priming the context area, and the task context switch extension would save and restore the context of the device.

For more information on user extensions, refer to the User Extensions chapter.

19.5 Multiprocessor Communications Interface (MPCI)

RTEMS requires that an MPCI layer be provided when a multiple node application is developed. This MPCI layer must provide an efficient and reliable communications mechanism between the multiple nodes. Tasks on different nodes communicate and synchronize with one another via the MPCI. Each MPCI layer must be tailored to support the architecture of the target platform.

For more information on the MPCI, refer to the Multiprocessing Manager chapter.

19.5.1 Tightly-Coupled Systems

A tightly-coupled system is a multiprocessor configuration in which the processors communicate solely via shared global memory. The MPCI can simply place the RTEMS packets in the shared memory space. The two primary considerations when designing an MPCI for a tightly-coupled system are data consistency and informing another node of a packet.

The data consistency problem may be solved using atomic "test and set" operations to provide a "lock" in the shared memory. It is important to minimize the length of time any particular processor locks a shared data structure.

The problem of informing another node of a packet can be addressed using one of two techniques. The first technique is to use an interprocessor interrupt capability to cause an interrupt on the receiving node. This technique requires that special support hardware be provided by either the processor itself or the target platform. The second technique is to have a node poll for arrival of packets. The drawback to this technique is the overhead associated with polling.

19.5.2 Loosely-Coupled Systems

A loosely-coupled system is a multiprocessor configuration in which the processors communicate via some type of communications link which is not shared global memory. The MPCI sends the RTEMS packets across the communications link to the destination node. The characteristics of the communications link vary widely and have a significant impact on the MPCI layer. For example, the bandwidth of the communications link has an obvious impact on the maximum MPCI throughput.

The characteristics of a shared network, such as Ethernet, lend themselves to supporting an MPCI layer. These networks provide both the point-to-point and broadcast capabilities which are expected by RTEMS.

19.5.3 Systems with Mixed Coupling

A mixed-coupling system is a multiprocessor configuration in which the processors communicate via both shared memory and communications links. A unique characteristic of mixed-coupling systems is that a node may not have access to all communication methods. There may be multiple shared memory areas and communication links. Therefore, one of the primary functions of the MPCI layer is to efficiently route RTEMS packets between nodes. This routing may be based on numerous algorithms. In addition, the router may provide alternate communications paths in the event of an overload or a partial failure.

19.5.4 Heterogeneous Systems

Designing an MPCI layer for a heterogeneous system requires special considerations by the developer. RTEMS is designed to eliminate many of the problems associated with sharing data in a heterogeneous environment. The MPCI layer need only address the representation of thirty-two (32) bit unsigned quantities.

For more information on supporting a heterogeneous system, refer the Supporting Heterogeneous Environments in the Multiprocessing Manager chapter.

20 User Extensions Manager

20.1 Introduction

The RTEMS User Extensions Manager allows the application developer to augment the executive by allowing them to supply extension routines which are invoked at critical system events. The directives provided by the user extensions manager are:

- `rtems_extension_create` - Create an extension set
- `rtems_extension_ident` - Get ID of an extension set
- `rtems_extension_delete` - Delete an extension set

20.2 Background

User extension routines are invoked when the following system events occur:

- Task creation
- Task initiation
- Task reinitiation
- Task deletion
- Task context switch
- Post task context switch
- Task begin
- Task exits
- Fatal error detection

These extensions are invoked as a function with arguments that are appropriate to the system event.

20.2.1 Extension Sets

An extension set is defined as a set of routines which are invoked at each of the critical system events at which user extension routines are invoked. Together a set of these routines typically perform a specific functionality such as performance monitoring or debugger support. RTEMS is informed of the entry points which constitute an extension set via the following structure:

```
typedef struct {
  User_extensions_thread_create_extension      thread_create;
  User_extensions_thread_start_extension       thread_start;
  User_extensions_thread_restart_extension     thread_restart;
  User_extensions_thread_delete_extension      thread_delete;
  User_extensions_thread_switch_extension      thread_switch;
  User_extensions_thread_post_switch_extension thread_post_switch;
  User_extensions_thread_begin_extension       thread_begin;
  User_extensions_thread_exitted_extension     thread_exitted;
  User_extensions_fatal_error_extension        fatal;
} User_extensions_Table;
```

RTEMS allows the user to have multiple extension sets active at the same time. First, a single static extension set may be defined as the application's User Extension Table which is included as part of the Configuration Table. This extension set is active for the entire life of the system and may not be deleted. This extension set is especially important because it is the only way the application can provided a FATAL error extension which is invoked if RTEMS fails during the initialize_executive directive. The static extension set is optional and may be configured as NULL if no static extension set is required.

Second, the user can install dynamic extensions using the `rtems_extension_create` directive. These extensions are RTEMS objects in that they have a name, an ID, and can be dynamically created and deleted. In contrast to the static extension set, these extensions can only be created and installed after the initialize_executive directive successfully completes execution. Dynamic extensions are useful for encapsulating the functionality of an extension set. For example, the application could use extensions to manage a special coprocessor, do performance monitoring, and to do stack bounds checking. Each of these extension sets could be written and installed independently of the others.

All user extensions are optional and RTEMS places no naming restrictions on the user.

20.2.2 TCB Extension Area

RTEMS provides for a pointer to a user-defined data area for each extension set to be linked to each task's control block. This set of pointers is an extension of the TCB and can be used to store additional data required by the user's extension functions. It is also possible for a user extension to utilize the notepad locations associated with each task although this may conflict with application usage of those particular notepads.

The TCB extension is an array of pointers in the TCB. The number of pointers in the area is the same as the number of user extension sets configured. This allows an application to augment the TCB with user-defined information. For example, an application could implement task profiling by storing timing statistics in the TCB's extended memory area. When a task context switch is being executed, the TASK_SWITCH extension could read a real-time clock to calculate how long the task being swapped out has run as well as timestamp the starting time for the task being swapped in.

If used, the extended memory area for the TCB should be allocated and the TCB extension pointer should be set at the time the task is created or started by either the TASK_CREATE or TASK_START extension. The application is responsible for managing this extended memory area for the TCBs. The memory may be reinitialized by the TASK_RESTART extension and should be deallocated by the TASK_DELETE extension when the task is deleted. Since the TCB extension buffers would most likely be of a fixed size, the RTEMS partition manager could be used to manage the application's extended memory area. The application could create a partition of fixed size TCB extension buffers and use the partition manager's allocation and deallocation directives to obtain and release the extension buffers.

20.2.3 Extensions

The sections that follow will contain a description of each extension. Each section will contain a prototype of a function with the appropriate calling sequence for the corresponding extension. The names given for the C function and its arguments are all defined by the user. The names used in the examples were arbitrarily chosen and impose no naming conventions on the user.

20.2.4 TASK_CREATE Extension

The TASK_CREATE extension directly corresponds to the task_create directive. If this extension is defined in any static or dynamic extension set and a task is being created, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
rtems_extension user_task_create(
  rtems_tcb *current_task,
  rtems_tcb *new_task
);
```

where current_task can be used to access the TCB for the currently executing task, and new_task can be used to access the TCB for the new task being created. This extension is invoked from the task_create directive after new_task has been completely initialized, but before it is placed on a ready TCB chain.

20.2.5 TASK_START Extension

The TASK_START extension directly corresponds to the task_start directive. If this extension is defined in any static or dynamic extension set and a task is being started, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
rtems_extension user_task_start(
  rtems_tcb *current_task,
  rtems_tcb *started_task
);
```

where current_task can be used to access the TCB for the currently executing task, and started_task can be used to access the TCB for the dormant task being started. This extension is invoked from the task_start directive after started_task has been made ready to start execution, but before it is placed on a ready TCB chain.

20.2.6 TASK_RESTART Extension

The TASK_RESTART extension directly corresponds to the task_restart directive. If this extension is defined in any static or dynamic extension set and a task is being restarted, then the extension should have a prototype similar to the following:

```
rtems_extension user_task_restart(
  rtems_tcb *current_task,
  rtems_tcb *restarted_task
);
```

where current_task can be used to access the TCB for the currently executing task, and restarted_task can be used to access the TCB for the task being restarted. This extension is invoked from the task_restart directive after restarted_task has been made ready to start execution, but before it is placed on a ready TCB chain.

20.2.7 TASK_DELETE Extension

The TASK_DELETE extension is associated with the task_delete directive. If this extension is defined in any static or dynamic extension set and a task is being deleted, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
rtems_extension user_task_delete(
  rtems_tcb *current_task,
  rtems_tcb *deleted_task
);
```

where current_task can be used to access the TCB for the currently executing task, and deleted_task can be used to access the TCB for the task being deleted. This extension is invoked from the task_delete directive after the TCB has been removed from a ready TCB chain, but before all its resources including the TCB have been returned to their respective free pools. This extension should not call any RTEMS directives if a task is deleting itself (current_task is equal to deleted_task).

20.2.8 TASK_SWITCH Extension

The TASK_SWITCH extension corresponds to a task context switch. If this extension is defined in any static or dynamic extension set and a task context switch is in progress, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
rtems_extension user_task_switch(
  rtems_tcb *current_task,
  rtems_tcb *heir_task
);
```

where current_task can be used to access the TCB for the task that is being swapped out, and heir_task can be used to access the TCB for the task being swapped in. This extension is invoked from RTEMS' dispatcher routine after the current_task context has been saved, but before the heir_task context has been restored. This extension should not call any RTEMS directives.

20.2.9 TASK_POST_SWITCH Extension

The TASK_POST_SWITCH extension corresponds to a task context switch. If this extension is defined in any static or dynamic extension set and a raw task context switch has been completed, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
rtems_extension user_task_post_switch(
  rtems_tcb *current_task
);
```

where current_task can be used to access the TCB for the task that is being swapped out, and heir_task can be used to access the TCB for the task being swapped in. This extension is invoked from RTEMS' dispatcher routine after the current_task context has been restored and the extension runs in the context of the current_task.

20.2.10 TASK_BEGIN Extension

The TASK_BEGIN extension is invoked when a task begins execution. It is invoked immediately before the body of the starting procedure and executes in the context in the task. This user extension have a prototype similar to the following:

```
rtems_extension user_task_begin(
  rtems_tcb *current_task
);
```

where current_task can be used to access the TCB for the currently executing task which has begun. The distinction between the TASK_BEGIN and TASK_START extension is that the TASK_BEGIN extension is executed in the context of the actual task while the TASK_START extension is executed in the context of the task performing the task_start directive. For most extensions, this is not a critical distinction.

20.2.11 TASK_EXITTED Extension

The TASK_EXITTED extension is invoked when a task exits the body of the starting procedure by either an implicit or explicit return statement. This user extension have a prototype similar to the following:

```
rtems_extension user_task_exitted(
  rtems_tcb *current_task
);
```

where current_task can be used to access the TCB for the currently executing task which has just exitted.

Although exiting of task is often considered to be a fatal error, this extension allows recovery by either restarting or deleting the exiting task. If the user does not wish to recover, then a fatal error may be reported. If the user does not provide a TASK_EXITTED extension or the provided handler returns control to RTEMS, then the RTEMS default handler will be used. This default handler invokes the directive fatal_error_occurred with the `RTEMS_TASK_EXITTED` directive status.

20.2.11.1 FATAL Error Extension

The FATAL error extension is associated with the fatal_error_occurred directive. If this extension is defined in any static or dynamic extension set and the fatal_error_occurred directive has been invoked, then this extension will be called. This extension should have a prototype similar to the following:

```
rtems_extension user_fatal_error(
  Internal_errors_Source  the_source,
  rtems_boolean           is_internal,
  rtems_unsigned32        the_error
);
```

where the_error is the error code passed to the fatal_error_occurred directive. This extension is invoked from the fatal_error_occurred directive.

If defined, the user's FATAL error extension is invoked before RTEMS' default fatal error routine is invoked and the processor is stopped. For example, this extension could be used to pass control to a debugger when a fatal error occurs. This extension should not call any RTEMS directives.

20.2.12 Order of Invocation

When one of the critical system events occur, the user extensions are invoked in either "forward" or "reverse" order. Forward order indicates that the static extension set is invoked followed by the dynamic extension sets in the order in which they were created. Reverse order means that the dynamic extension sets are invoked in the opposite of the order in which they were created followed by the static extension set. By invoking the extension sets in this order, extensions can be built upon one another. At the following system events, the extensions are invoked in forward order:

- Task creation
- Task initiation
- Task reinitiation
- Task deletion
- Task context switch
- Post task context switch
- Task begins to execute

At the following system events, the extensions are invoked in reverse order:

- Task deletion
- Fatal error detection

At these system events, the extensions are invoked in reverse order to insure that if an extension set is built upon another, the more complicated extension is invoked before the extension set it is built upon. For example, by invoking the static extension set last it is known that the "system" fatal error extension will be the last fatal error extension executed. Another example is use of the task delete extension by the Standard C Library. Extension sets which are installed after the Standard C Library will operate correctly even if they utilize the C Library because the C Library's TASK_DELETE extension is invoked after that of the other extensions.

20.3 Operations

20.3.1 Creating an Extension Set

The `rtems_extension_create` directive creates and installs an extension set by allocating a Extension Set Control Block (ESCB), assigning the extension set a user-specified name, and assigning it an extension set ID. Newly created extension sets are immediately installed and are invoked upon the next system even supporting an extension.

20.3.2 Obtaining Extension Set IDs

When an extension set is created, RTEMS generates a unique extension set ID and assigns it to the created extension set until it is deleted. The extension ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_extension_create` directive, the extension set ID is stored in a user provided location. Second, the extension set ID may be obtained later using the `rtems_extension_ident` directive. The extension set ID is used by other directives to manipulate this extension set.

20.3.3 Deleting an Extension Set

The `rtems_extension_delete` directive is used to delete an extension set. The extension set's control block is returned to the ESCB free list when it is deleted. An extension set can be deleted by a task other than the task which created the extension set. Any subsequent references to the extension's name and ID are invalid.

20.4 Directives

This section details the user extension manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

20.4.1 EXTENSION_CREATE - Create a extension set

CALLING SEQUENCE:

```
rtems_status_code rtems_extension_create(
  rtems_name              name,
  rtems_extensions_table *table,
  rtems_id               *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - extension set created successfully
`RTEMS_INVALID_NAME` - invalid extension set name
`RTEMS_TOO_MANY` - too many extension sets created

DESCRIPTION:

This directive creates a extension set. The assigned extension set id is returned in id. This id is used to access the extension set with other user extension manager directives. For control and maintenance of the extension set, RTEMS allocates an ESCB from the local ESCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

20.4.2 EXTENSION_IDENT - Get ID of a extension set

CALLING SEQUENCE:

```
rtems_status_code rtems_extension_ident(
  rtems_name  name,
  rtems_id   *id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - extension set identified successfully
`RTEMS_INVALID_NAME` - extension set name not found

DESCRIPTION:

This directive obtains the extension set id associated with the extension set name to be acquired. If the extension set name is not unique, then the extension set id will match one of the extension sets with that name. However, this extension set id is not guaranteed to correspond to the desired extension set. The extension set id is used to access this extension set in other extension set related directives.

NOTES:

This directive will not cause the running task to be preempted.

20.4.3 EXTENSION_DELETE - Delete a extension set

CALLING SEQUENCE:

```
rtems_status_code rtems_extension_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

`RTEMS_SUCCESSFUL` - extension set deleted successfully
`RTEMS_INVALID_ID` - invalid extension set id

DESCRIPTION:

This directive deletes the extension set specified by id. If the extension set is running, it is automatically canceled. The ESCB for the deleted extension set is reclaimed by RTEMS.

NOTES:

This directive will not cause the running task to be preempted.

A extension set can be deleted by a task other than the task which created the extension set.

NOTES:

This directive will not cause the running task to be preempted.

21 Configuring a System

21.1 Configuration Table

The RTEMS Configuration Table is used to tailor an application for its specific needs. For example, the user can configure the number of device drivers or which APIs may be used. THe address of the user-defined Configuration Table is passed as an argument to the `rtems_initialize_executive` directive, which MUST be the first RTEMS directive called. The RTEMS Configuration Table is defined in the following C structure:

```
typedef struct {
  void                             *work_space_start;
  rtems_unsigned32                  work_space_size;
  rtems_unsigned32                  maximum_extensions;
  rtems_unsigned32                  microseconds_per_tick;
  rtems_unsigned32                  ticks_per_timeslice;
  rtems_unsigned32                  maximum_devices;
  rtems_unsigned32                  number_of_device_drivers;
  rtems_driver_address_table       *Device_driver_table;
  rtems_unsigned32                  number_of_initial_extensions;
  rtems_extensions_table           *User_extension_table;
  rtems_multiprocessing_table      *User_multiprocessing_table;
  rtems_api_configuration_table    *RTEMS_api_configuration;
  posix_api_configuration_table    *POSIX_api_configuration;
} rtems_configuration_table;
```

work_space_start
is the address of the RTEMS RAM Workspace. This area contains items such as the various object control blocks (TCBs, QCBs, ...) and task stacks. If the address is not aligned on a four-word boundary, then RTEMS will invoke the fatal error handler during `rtems_initialize_executive`.

work_space_size
is the calculated size of the RTEMS RAM Workspace. The section Sizing the RTEMS RAM Workspace details how to arrive at this number.

microseconds_per_tick
is number of microseconds per clock tick.

ticks_per_timeslice
is the number of clock ticks for a timeslice.

maximum_devices
is the maximum number of devices that can be registered.

number_of_device_drivers
is the number of device drivers for the system. There should be the same number of entries in the Device Driver Table. If this field is zero, then the User_driver_address_table entry should be NULL.

Device_driver_table
is the address of the Device Driver Table. This table contains the entry points for each device driver. If the number_of_device_drivers field is zero,

then this entry should be NULL. The format of this table will be discussed below.

number_of_initial_extensions
is the number of initial user extensions. There should be the same number of entries as in the User_extension_table. If this field is zero, then the User_driver_address_table entry should be NULL.

User_extension_table is the address of the User Extension Table. This table contains the entry points for the static set of optional user extensions. If no user extensions are configured, then this entry should be NULL. The format of this table will be discussed below.

User_multiprocessing_table
is the address of the Multiprocessor Configuration Table. This table contains information needed by RTEMS only when used in a multiprocessor configuration. This field must be NULL when RTEMS is used in a single processor configuration.

RTEMS_api_configuration
is the address of the RTEMS API Configuration Table. This table contains information needed by the RTEMS API. This field should be NULL if the RTEMS API is not used. [NOTE: Currently the RTEMS API is required to support support components such as BSPs and libraries which use this API.]

POSIX_api_configuration
is the address of the POSIX API Configuration Table. This table contains information needed by the POSIX API. This field should be NULL if the POSIX API is not used.

21.2 RTEMS API Configuration Table

The RTEMS API Configuration Table is used to configure the managers which constitute the RTEMS API for a particular application. For example, the user can configure the maximum number of tasks for this application. The RTEMS API Configuration Table is defined in the following C structure:

```
typedef struct {
  rtems_unsigned32                  maximum_tasks;
  rtems_unsigned32                  maximum_timers;
  rtems_unsigned32                  maximum_semaphores;
  rtems_unsigned32                  maximum_message_queues;
  rtems_unsigned32                  maximum_partitions;
  rtems_unsigned32                  maximum_regions;
  rtems_unsigned32                  maximum_ports;
  rtems_unsigned32                  maximum_periods;
  rtems_unsigned32                  number_of_initialization_tasks;
  rtems_initialization_tasks_table *User_initialization_tasks_table;
} rtems_api_configuration_table;
```

maximum_tasks	is the maximum number of tasks that can be concurrently active (created) in the system including initialization tasks.
maximum_timers	is the maximum number of timers that can be concurrently active in the system.
maximum_semaphores	is the maximum number of semaphores that can be concurrently active in the system.
maximum_message_queues	is the maximum number of message queues that can be concurrently active in the system.
maximum_partitions	is the maximum number of partitions that can be concurrently active in the system.
maximum_regions	is the maximum number of regions that can be concurrently active in the system.
maximum_ports	is the maximum number of ports into dual-port memory areas that can be concurrently active in the system.
number_of_initialization_tasks	is the number of initialization tasks configured. At least one initialization task must be configured.
User_initialization_tasks_table	is the address of the Initialization Task Table. This table contains the information needed to create and start each of the initialization tasks. The format of this table will be discussed below.

21.3 POSIX API Configuration Table

The POSIX API Configuration Table is used to configure the managers which constitute the POSIX API for a particular application. For example, the user can configure the maximum number of threads for this application. The POSIX API Configuration Table is defined in the following C structure:

```
typedef struct {
  void       *(*thread_entry)(void *);
} posix_initialization_threads_table;

typedef struct {
  int                                   maximum_threads;
  int                                   maximum_mutexes;
  int                                   maximum_condition_variables;
  int                                   maximum_keys;
  int                                   maximum_queued_signals;
  int                                   number_of_initialization_tasks;
  posix_initialization_threads_table *User_initialization_tasks_table;
} posix_api_configuration_table;
```

maximum_threads is the maximum number of threads that can be concurrently active (created) in the system including initialization threads.

maximum_mutexes is the maximum number of mutexes that can be concurrently active in the system.

maximum_condition_variables
is the maximum number of condition variables that can be concurrently active in the system.

maximum_keys is the maximum number of keys that can be concurrently active in the system.

maximum_queued_signals
is the maximum number of queued signals that can be concurrently pending in the system.

number_of_initialization_threads
is the number of initialization threads configured. At least one initialization threads must be configured.

User_initialization_threads_table
is the address of the Initialization Threads Table. This table contains the information needed to create and start each of the initialization threads. The format of each entry in this table is defined in the posix_initialization_threads_table structure.

21.4 CPU Dependent Information Table

The CPU Dependent Information Table is used to describe processor dependent information required by RTEMS. This table is generally used to supply RTEMS with information only known by the Board Support Package. The contents of this table are discussed in the CPU Dependent Information Table chapter of the Applications Supplement document for a specific target processor.

21.5 Initialization Task Table

The Initialization Task Table is used to describe each of the user initialization tasks to the Initialization Manager. The table contains one entry for each initialization task the user wishes to create and start. The fields of this data structure directly correspond to arguments to the task_create and task_start directives. The number of entries is found in the number_of_initialization_tasks entry in the Configuration Table. The format of each entry in the Initialization Task Table is defined in the following C structure:

```
typedef struct {
  rtems_name           name;
  rtems_unsigned32     stack_size;
  rtems_task_priority  initial_priority;
  rtems_attribute      attribute_set;
  rtems_task_entry     entry_point;
  rtems_mode           mode_set;
  rtems_task_argument  argument;
} rtems_initialization_tasks_table;
```

name	is the name of this initialization task.
stack_size	is the size of the stack for this initialization task.
initial_priority	is the priority of this initialization task.
attribute_set	is the attribute set used during creation of this initialization task.
entry_point	is the address of the entry point of this initialization task.
mode_set	is the initial execution mode of this initialization task.
argument	is the initial argument for this initialization task.

A typical declaration for an Initialization Task Table might appear as follows:

```
rtems_initialization_tasks_table
Initialization_tasks[2] = {
   { INIT_1_NAME,
     1024,
     1,
     DEFAULT_ATTRIBUTES,
     Init_1,
     DEFAULT_MODES,
     1

   },
   { INIT_2_NAME,
        1024,
        250,
        FLOATING_POINT,
        Init_2,
        NO_PREEMPT,
        2

      }
    };
```

21.6 Driver Address Table

The Device Driver Table is used to inform the I/O Manager of the set of entry points for each device driver configured in the system. The table contains one entry for each device driver required by the application. The number of entries is defined in the number_of_device_drivers entry in the Configuration Table. The format of each entry in the Device Driver Table is defined in the following C structure:

```
typedef struct {
  rtems_device_driver_entry initialization;
  rtems_device_driver_entry open;
  rtems_device_driver_entry close;
  rtems_device_driver_entry read;
  rtems_device_driver_entry write;
  rtems_device_driver_entry control;
} rtems_driver_address_table;
```

initialization	is the address of the entry point called by `rtems_io_initialize` to initialize a device driver and its associated devices.
open	is the address of the entry point called by `rtems_io_open`.
close	is the address of the entry point called by `rtems_io_close`.
read	is the address of the entry point called by `rtems_io_read`.
write	is the address of the entry point called by `rtems_io_write`.
control	is the address of the entry point called by `rtems_io_control`.

Driver entry points configured as NULL will always return a status code of `RTEMS_SUCCESSFUL`. No user code will be executed in this situation.

A typical declaration for a Device Driver Table might appear as follows:

```
rtems_driver_address_table Driver_table[2] = {
   { tty_initialize, tty_open,  tty_close,  /* major = 0 */
     tty_read,       tty_write, tty_control
   },
   { lp_initialize, lp_open,    lp_close,   /* major = 1 */
```

```
      NULL,          lp_write,   lp_control
    }
};
```

More information regarding the construction and operation of device drivers is provided in the I/O Manager chapter.

21.7 User Extensions Table

The User Extensions Table is used to inform RTEMS of the optional user-supplied static extension set. This table contains one entry for each possible extension. The entries are called at critical times in the life of the system and individual tasks. The application may create dynamic extensions in addition to this single static set. The format of each entry in the User Extensions Table is defined in the following C structure:

```
typedef User_extensions_routine                   rtems_extension;
typedef User_extensions_thread_create_extension   rtems_task_create_extension;
typedef User_extensions_thread_delete_extension   rtems_task_delete_extension;
typedef User_extensions_thread_start_extension    rtems_task_start_extension;
typedef User_extensions_thread_restart_extension  rtems_task_restart_extension;
typedef User_extensions_thread_switch_extension   rtems_task_switch_extension;
typedef User_extensions_thread_begin_extension    rtems_task_begin_extension;
typedef User_extensions_thread_exitted_extension  rtems_task_exitted_extension;
typedef User_extensions_fatal_extension           rtems_fatal_extension;

typedef User_extensions_Table                     rtems_extensions_table;

typedef struct {
  rtems_task_create_extension      thread_create;
  rtems_task_start_extension       thread_start;
  rtems_task_restart_extension     thread_restart;
  rtems_task_delete_extension      thread_delete;
  rtems_task_switch_extension      thread_switch;
  rtems_task_post_switch_extension thread_post_switch;
  rtems_task_begin_extension       thread_begin;
  rtems_task_exitted_extension     thread_exitted;
  rtems_fatal_extension            fatal;
} User_extensions_Table;
```

thread_create is the address of the user-supplied subroutine for the TASK_CREATE extension. If this extension for task creation is defined, it is called from the task_create directive. A value of NULL indicates that no extension is provided.

thread_start is the address of the user-supplied subroutine for the TASK_START extension. If this extension for task initiation is defined, it is called from the task_start directive. A value of NULL indicates that no extension is provided.

thread_restart	is the address of the user-supplied subroutine for the TASK_RESTART extension. If this extension for task re-initiation is defined, it is called from the task_restart directive. A value of NULL indicates that no extension is provided.
thread_delete	is the address of the user-supplied subroutine for the TASK_DELETE extension. If this RTEMS extension for task deletion is defined, it is called from the task_delete directive. A value of NULL indicates that no extension is provided.
thread_switch	is the address of the user-supplied subroutine for the task context switch extension. This subroutine is called from RTEMS dispatcher after the current task has been swapped out but before the new task has been swapped in. A value of NULL indicates that no extension is provided. As this routine is invoked after saving the current task's context and before restoring the heir task's context, it is not necessary for this routine to save and restore any registers.
thread_post_switch	is the address of the user-supplied subroutine for the post task context switch extension. This subroutine is called from RTEMS dispatcher in the context of the task which has just been swapped in.
thread_begin	is the address of the user-supplied subroutine which is invoked immediately before a task begins execution. It is invoked in the context of the beginning task. A value of NULL indicates that no extension is provided.
thread_exitted	is the address of the user-supplied subroutine which is invoked when a task exits. This procedure is responsible for some action which will allow the system to continue execution (i.e. delete or restart the task) or to terminate with a fatal error. If this field is set to NULL, the default RTEMS TASK_EXITTED handler will be invoked.
fatal	is the address of the user-supplied subroutine for the FATAL extension. This RTEMS extension of fatal error handling is called from the `rtems_fatal_error_occurred` directive. If the user's fatal error handler returns or if this entry is NULL then the default RTEMS fatal error handler will be executed.

A typical declaration for a User Extension Table which defines the TASK_CREATE, TASK_DELETE, TASK_SWITCH, and FATAL extension might appear as follows:

```
rtems_extensions_table User_extensions = {
   task_create_extension,
   NULL,
   NULL,
    task_delete_extension,
    task_switch_extension,
    NULL,
    NULL,
    fatal_extension
};
```

More information regarding the user extensions is provided in the User Extensions chapter.

21.8 Multiprocessor Configuration Table

The Multiprocessor Configuration Table contains information needed when using RTEMS in a multiprocessor configuration. Many of the details associated with configuring a multiprocessor system are dependent on the multiprocessor communications layer provided by the user. The address of the Multiprocessor Configuration Table should be placed in the User_multiprocessing_table entry in the primary Configuration Table. Further details regarding many of the entries in the Multiprocessor Configuration Table will be provided in the Multiprocessing chapter. The format of the Multiprocessor Configuration Table is defined in the following C structure:

```
typedef struct {
  rtems_unsigned32  node;
  rtems_unsigned32  maximum_nodes;
  rtems_unsigned32  maximum_global_objects;
  rtems_unsigned32  maximum_proxies;
  rtems_mpci_table *User_mpci_table;
} rtems_multiprocessing_table;
```

node is a unique processor identifier and is used in routing messages between nodes in a multiprocessor configuration. Each processor must have a unique node number. RTEMS assumes that node numbers start at one and increase sequentially. This assumption can be used to advantage by the user-supplied MPCI layer. Typically, this requirement is made when the node numbers are used to calculate the address of inter-processor communication links. Zero should be avoided as a node number because some MPCI layers use node zero to represent broadcasted packets. Thus, it is recommended that node numbers start at one and increase sequentially.

maximum_nodes is the number of processor nodes in the system.

maximum_global_objects
is the maximum number of global objects which can exist at any given moment in the entire system. If this parameter is not the same on all nodes in the system, then a fatal error is generated to inform the user that the system is inconsistent.

maximum_proxies is the maximum number of proxies which can exist at any given moment on this particular node. A proxy is a substitute task control block which represent a task residing on a remote node when that task blocks on a remote object. Proxies are used in situations in which delayed interaction is required with a remote node.

User_mpci_table is the address of the Multiprocessor Communications Interface Table. This table contains the entry points of user-provided functions which constitute the multiprocessor communications layer. This table must be provided in multiprocessor configurations with all entries configured. The format of this table and details regarding its entries can be found in the next section.

21.9 Multiprocessor Communications Interface Table

The format of this table is defined in the following C structure:

```
typedef struct {
  rtems_unsigned32                  default_timeout; /* in ticks */
  rtems_unsigned32                  maximum_packet_size;
  rtems_mpci_initialization_entry initialization;
  rtems_mpci_get_packet_entry     get_packet;
  rtems_mpci_return_packet_entry  return_packet;
  rtems_mpci_send_entry           send;
  rtems_mpci_receive_entry        receive;
} rtems_mpci_table;
```

default_timeout is the default maximum length of time a task should block waiting for a response to a directive which results in communication with a remote node. The maximum length of time is a function the user supplied multiprocessor communications layer and the media used. This timeout only applies to directives which would not block if the operation were performed locally.

maximum_packet_size
is the size in bytes of the longest packet which the MPCI layer is capable of sending. This value should represent the total number of bytes available for a RTEMS interprocessor messages.

initialization is the address of the entry point for the initialization procedure of the user supplied multiprocessor communications layer.

get_packet is the address of the entry point for the procedure called by RTEMS to obtain a packet from the user supplied multiprocessor communications layer.

return_packet is the address of the entry point for the procedure called by RTEMS to return a packet to the user supplied multiprocessor communications layer.

send is the address of the entry point for the procedure called by RTEMS to send an envelope to another node. This procedure is part of the user supplied multiprocessor communications layer.

receive is the address of the entry point for the procedure called by RTEMS to retrieve an envelope containing a message from another node. This procedure is part of the user supplied multiprocessor communications layer.

More information regarding the required functionality of these entry points is provided in the Multiprocessor chapter.

21.10 Determining Memory Requirements

Since memory is a critical resource in many real-time embedded systems, RTEMS was specifically designed to allow unused managers to be excluded from the run-time environment. This allows the application designer the flexibility to tailor RTEMS to most efficiently meet system requirements while still satisfying even the most stringent memory constraints. As result, the size of the RTEMS executive is application dependent. A Memory Requirements worksheet is provided in the Applications Supplement document for a specific target processor. This worksheet can be used to calculate the memory requirements of a custom RTEMS run-time environment. To insure that enough memory is allocated for future versions of RTEMS, the application designer should round these memory requirements up. The following managers may be optionally excluded:

- signal
- region
- dual ported memory
- event
- multiprocessing
- partition
- timer
- semaphore
- message
- rate monotonic

RTEMS based applications must somehow provide memory for RTEMS' code and data space. Although RTEMS' data space must be in RAM, its code space can be located in either ROM or RAM. In addition, the user must allocate RAM for the RTEMS RAM Workspace. The size of this area is application dependent and can be calculated using the formula provided in the Memory Requirements chapter of the Applications Supplement document for a specific target processor.

All RTEMS data variables and routine names used by RTEMS begin with the underscore (_) character followed by an upper-case letter. If RTEMS is linked with an application, then the application code should NOT contain any symbols which begin with the underscore character and followed by an upper-case letter to avoid any naming conflicts. All RTEMS directive names should be treated as reserved words.

21.11 Sizing the RTEMS RAM Workspace

The RTEMS RAM Workspace is a user-specified block of memory reserved for use by RTEMS. The application should NOT modify this memory. This area consists primarily of the RTEMS data structures whose exact size depends upon the values specified in the Configuration Table. In addition, task stacks and floating point context areas are dynamically allocated from the RTEMS RAM Workspace.

The starting address of the RTEMS RAM Workspace must be aligned on a four-byte boundary. Failure to properly align the workspace area will result in the `rtems_fatal_error_occurred` directive being invoked with the `RTEMS_INVALID_ADDRESS` error code.

A worksheet is provided in the Memory Requirements chapter of the Applications Supplement document for a specific target processor to assist the user in calculating the minimum size of the RTEMS RAM Workspace for each application. The value calculated with this worksheet is the minimum value that should be specified as the work_space_size parameter of the Configuration Table. The user is cautioned that future versions of RTEMS may not have the same memory requirements per object. Although the value calculated is sufficient for a particular target processor and release of RTEMS, memory usage is subject to change across versions and target processors. The user is advised to allocate somewhat more memory than the worksheet recommends to insure compatibility with future releases for a specific target processor and other target processors. To avoid problems, the user should recalculate the memory requirements each time one of the following events occurs:

- a configuration parameter is modified,
- task or interrupt stack requirements change,
- task floating point attribute is altered,
- RTEMS is upgraded, or
- the target processor is changed.

Failure to provide enough space in the RTEMS RAM Workspace will result in the `rtems_fatal_error_occurred` directive being invoked with the appropriate error code.

22 Multiprocessing Manager

22.1 Introduction

In multiprocessor real-time systems, new requirements, such as sharing data and global resources between processors, are introduced. This requires an efficient and reliable communications vehicle which allows all processors to communicate with each other as necessary. In addition, the ramifications of multiple processors affect each and every characteristic of a real-time system, almost always making them more complicated.

RTEMS addresses these issues by providing simple and flexible real-time multiprocessing capabilities. The executive easily lends itself to both tightly-coupled and loosely-coupled configurations of the target system hardware. In addition, RTEMS supports systems composed of both homogeneous and heterogeneous mixtures of processors and target boards.

A major design goal of the RTEMS executive was to transcend the physical boundaries of the target hardware configuration. This goal is achieved by presenting the application software with a logical view of the target system where the boundaries between processor nodes are transparent. As a result, the application developer may designate objects such as tasks, queues, events, signals, semaphores, and memory blocks as global objects. These global objects may then be accessed by any task regardless of the physical location of the object and the accessing task. RTEMS automatically determines that the object being accessed resides on another processor and performs the actions required to access the desired object. Simply stated, RTEMS allows the entire system, both hardware and software, to be viewed logically as a single system.

22.2 Background

RTEMS makes no assumptions regarding the connection media or topology of a multiprocessor system. The tasks which compose a particular application can be spread among as many processors as needed to satisfy the application's timing requirements. The application tasks can interact using a subset of the RTEMS directives as if they were on the same processor. These directives allow application tasks to exchange data, communicate, and synchronize regardless of which processor they reside upon.

The RTEMS multiprocessor execution model is multiple instruction streams with multiple data streams (MIMD). This execution model has each of the processors executing code independent of the other processors. Because of this parallelism, the application designer can more easily guarantee deterministic behavior.

By supporting heterogeneous environments, RTEMS allows the systems designer to select the most efficient processor for each subsystem of the application. Configuring RTEMS for a heterogeneous environment is no more difficult than for a homogeneous one. In keeping with RTEMS philosophy of providing transparent physical node boundaries, the minimal heterogeneous processing required is isolated in the MPCI layer.

22.2.1 Nodes

A processor in a RTEMS system is referred to as a node. Each node is assigned a unique non-zero node number by the application designer. RTEMS assumes that node numbers are assigned consecutively from one to maximum_nodes. The node number, node, and the maximum number of nodes, maximum_nodes, in a system are found in the Multiprocessor Configuration Table. The maximum_nodes field and the number of global objects, maximum_global_objects, is required to be the same on all nodes in a system.

The node number is used by RTEMS to identify each node when performing remote operations. Thus, the Multiprocessor Communications Interface Layer (MPCI) must be able to route messages based on the node number.

22.2.2 Global Objects

All RTEMS objects which are created with the GLOBAL attribute will be known on all other nodes. Global objects can be referenced from any node in the system, although certain directive specific restrictions (e.g. one cannot delete a remote object) may apply. A task does not have to be global to perform operations involving remote objects. The maximum number of global objects is the system is user configurable and can be found in the maximum_global_objects field in the Multiprocessor Configuration Table. The distribution of tasks to processors is performed during the application design phase. Dynamic task relocation is not supported by RTEMS.

22.2.3 Global Object Table

RTEMS maintains two tables containing object information on every node in a multiprocessor system: a local object table and a global object table. The local object table on each node is unique and contains information for all objects created on this node whether those objects are local or global. The global object table contains information regarding all global objects in the system and, consequently, is the same on every node.

Since each node must maintain an identical copy of the global object table, the maximum number of entries in each copy of the table must be the same. The maximum number of entries in each copy is determined by the maximum_global_objects parameter in the Multiprocessor Configuration Table. This parameter, as well as the maximum_nodes parameter, is required to be the same on all nodes. To maintain consistency among the table copies, every node in the system must be informed of the creation or deletion of a global object.

22.2.4 Remote Operations

When an application performs an operation on a remote global object, RTEMS must generate a Remote Request (RQ) message and send it to the appropriate node. After completing the requested operation, the remote node will build a Remote Response (RR) message and send it to the originating node. Messages generated as a side-effect of a directive (such as deleting a global task) are known as Remote Processes (RP) and do not require the receiving node to respond.

Other than taking slightly longer to execute directives on remote objects, the application is unaware of the location of the objects it acts upon. The exact amount of overhead required for a remote operation is dependent on the media connecting the nodes and, to a lesser degree, on the efficiency of the user-provided MPCI routines.

The following shows the typical transaction sequence during a remote application:

1. The application issues a directive accessing a remote global object.
2. RTEMS determines the node on which the object resides.
3. RTEMS calls the user-provided MPCI routine GET_PACKET to obtain a packet in which to build a RQ message.
4. After building a message packet, RTEMS calls the user-provided MPCI routine SEND_PACKET to transmit the packet to the node on which the object resides (referred to as the destination node).
5. The calling task is blocked until the RR message arrives, and control of the processor is transferred to another task.
6. The MPCI layer on the destination node senses the arrival of a packet (commonly in an ISR), and calls the `rtems_multiprocessing_announce` directive. This directive readies the Multiprocessing Server.
7. The Multiprocessing Server calls the user-provided MPCI routine RECEIVE_PACKET, performs the requested operation, builds an RR message, and returns it to the originating node.
8. The MPCI layer on the originating node senses the arrival of a packet (typically via an interrupt), and calls the RTEMS `rtems_multiprocessing_announce` directive. This directive readies the Multiprocessing Server.
9. The Multiprocessing Server calls the user-provided MPCI routine RECEIVE_PACKET, readies the original requesting task, and blocks until another packet arrives. Control is transferred to the original task which then completes processing of the directive.

If an uncorrectable error occurs in the user-provided MPCI layer, the fatal error handler should be invoked. RTEMS assumes the reliable transmission and reception of messages by the MPCI and makes no attempt to detect or correct errors.

22.2.5 Proxies

A proxy is an RTEMS data structure which resides on a remote node and is used to represent a task which must block as part of a remote operation. This action can occur as part of the `rtems_semaphore_obtain` and `rtems_message_queue_receive` directives. If the object were local, the task's control block would be available for modification to indicate it was blocking on a message queue or semaphore. However, the task's control block resides only on the same node as the task. As a result, the remote node must allocate a proxy to represent the task until it can be readied.

The maximum number of proxies is defined in the Multiprocessor Configuration Table. Each node in a multiprocessor system may require a different number of proxies to be configured. The distribution of proxy control blocks is application dependent and is different from the distribution of tasks.

22.2.6 Multiprocessor Configuration Table

The Multiprocessor Configuration Table contains information needed by RTEMS when used in a multiprocessor system. This table is discussed in detail in the section Multiprocessor Configuration Table of the Configuring a System chapter.

22.3 Multiprocessor Communications Interface Layer

The Multiprocessor Communications Interface Layer (MPCI) is a set of user-provided procedures which enable the nodes in a multiprocessor system to communicate with one another. These routines are invoked by RTEMS at various times in the preparation and processing of remote requests. Interrupts are enabled when an MPCI procedure is invoked. It is assumed that if the execution mode and/or interrupt level are altered by the MPCI layer, that they will be restored prior to returning to RTEMS.

The MPCI layer is responsible for managing a pool of buffers called packets and for sending these packets between system nodes. Packet buffers contain the messages sent between the nodes. Typically, the MPCI layer will encapsulate the packet within an envelope which contains the information needed by the MPCI layer. The number of packets available is dependent on the MPCI layer implementation.

The entry points to the routines in the user's MPCI layer should be placed in the Multiprocessor Communications Interface Table. The user must provide entry points for each of the following table entries in a multiprocessor system:

- initialization initialize the MPCI
- get_packet obtain a packet buffer
- return_packet return a packet buffer
- send_packet send a packet to another node

- receive_packet called to get an arrived packet

A packet is sent by RTEMS in each of the following situations:

- an RQ is generated on an originating node;
- an RR is generated on a destination node;
- a global object is created;
- a global object is deleted;
- a local task blocked on a remote object is deleted;
- during system initialization to check for system consistency.

If the target hardware supports it, the arrival of a packet at a node may generate an interrupt. Otherwise, the real-time clock ISR can check for the arrival of a packet. In any case, the `rtems_multiprocessing_announce` directive must be called to announce the arrival of a packet. After exiting the ISR, control will be passed to the Multiprocessing Server to process the packet. The Multiprocessing Server will call the get_packet entry to obtain a packet buffer and the receive_entry entry to copy the message into the buffer obtained.

22.3.1 INITIALIZATION

The INITIALIZATION component of the user-provided MPCI layer is called as part of the `rtems_initialize_executive` directive to initialize the MPCI layer and associated hardware. It is invoked immediately after all of the device drivers have been initialized. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_initialization(
  rtems_configuration_table *configuration
);
```

where configuration is the address of the user's Configuration Table. Operations on global objects cannot be performed until this component is invoked. The INITIALIZATION component is invoked only once in the life of any system. If the MPCI layer cannot be successfully initialized, the fatal error manager should be invoked by this routine.

One of the primary functions of the MPCI layer is to provide the executive with packet buffers. The INITIALIZATION routine must create and initialize a pool of packet buffers. There must be enough packet buffers so RTEMS can obtain one whenever needed.

22.3.2 GET_PACKET

The GET_PACKET component of the user-provided MPCI layer is called when RTEMS must obtain a packet buffer to send or broadcast a message. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_get_packet(
  rtems_packet_prefix **packet
);
```

where packet is the address of a pointer to a packet. This routine always succeeds and, upon return, packet will contain the address of a packet. If for any reason, a packet cannot be successfully obtained, then the fatal error manager should be invoked.

RTEMS has been optimized to avoid the need for obtaining a packet each time a message is sent or broadcast. For example, RTEMS sends response messages (RR) back to the originator in the same packet in which the request message (RQ) arrived.

22.3.3 RETURN_PACKET

The RETURN_PACKET component of the user-provided MPCI layer is called when RTEMS needs to release a packet to the free packet buffer pool. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_return_packet(
  rtems_packet_prefix *packet
);
```

where packet is the address of a packet. If the packet cannot be successfully returned, the fatal error manager should be invoked.

22.3.4 RECEIVE_PACKET

The RECEIVE_PACKET component of the user-provided MPCI layer is called when RTEMS needs to obtain a packet which has previously arrived. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_receive_packet(
  rtems_packet_prefix **packet
);
```

where packet is a pointer to the address of a packet to place the message from another node. If a message is available, then packet will contain the address of the message from another node. If no messages are available, this entry packet should contain NULL.

22.3.5 SEND_PACKET

The SEND_PACKET component of the user-provided MPCI layer is called when RTEMS needs to send a packet containing a message to another node. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_send_packet(
  rtems_unsigned32       node,
  rtems_packet_prefix  **packet
);
```

where node is the node number of the destination and packet is the address of a packet which containing a message. If the packet cannot be successfully sent, the fatal error manager should be invoked.

If node is set to zero, the packet is to be broadcasted to all other nodes in the system. Although some MPCI layers will be built upon hardware which support a broadcast mechanism, others may be required to generate a copy of the packet for each node in the system.

Many MPCI layers use the packet_length field of the MP_packet_prefix of the packet to avoid sending unnecessary data. This is especially useful if the media connecting the nodes is relatively slow.

The to_convert field of the MP_packet_prefix portion of the packet indicates how much of the packet (in unsigned32's) may require conversion in a heterogeneous system.

22.3.6 Supporting Heterogeneous Environments

Developing an MPCI layer for a heterogeneous system requires a thorough understanding of the differences between the processors which comprise the system. One difficult problem is the varying data representation schemes used by different processor types. The most pervasive data representation problem is the order of the bytes which compose a data entity. Processors which place the least significant byte at the smallest address are classified as little endian processors. Little endian byte-ordering is shown below:

```
+---------------+----------------+---------------+----------------+
|               |                |               |                |
|    Byte 3     |     Byte 2     |    Byte 1     |    Byte 0      |
|               |                |               |                |
+---------------+----------------+---------------+----------------+
```

Conversely, processors which place the most significant byte at the smallest address are classified as big endian processors. Big endian byte-ordering is shown below:

```
+---------------+----------------+---------------+----------------+
|               |                |               |                |
|    Byte 0     |     Byte 1     |    Byte 2     |    Byte 3      |
|               |                |               |                |
+---------------+----------------+---------------+----------------+
```

Unfortunately, sharing a data structure between big endian and little endian processors requires translation into a common endian format. An application designer typically chooses the common endian format to minimize conversion overhead.

Another issue in the design of shared data structures is the alignment of data structure elements. Alignment is both processor and compiler implementation dependent. For example, some processors allow data elements to begin on any address boundary, while others impose restrictions. Common restrictions are that data elements must begin on either an even address or on a long word boundary. Violation of these restrictions may cause an exception or impose a performance penalty.

Other issues which commonly impact the design of shared data structures include the representation of floating point numbers, bit fields, decimal data, and character strings. In addition, the representation method for negative integers could be one's or two's complement. These factors combine to increase the complexity of designing and manipulating data structures shared between processors.

RTEMS addressed these issues in the design of the packets used to communicate between nodes. The RTEMS packet format is designed to allow the MPCI layer to perform all necessary conversion without burdening the developer with the details of the RTEMS packet format. As a result, the MPCI layer must be aware of the following:

- All packets must begin on a four byte boundary.
- Packets are composed of both RTEMS and application data. All RTEMS data is treated as thirty-two (32) bit unsigned quantities and is in the first `RTEMS_MINIMUM_UNSIGNED32S_TO_CONVERT` thirty-two (32) quantities of the packet.
- The RTEMS data component of the packet must be in native endian format. Endian conversion may be performed by either the sending or receiving MPCI layer.
- RTEMS makes no assumptions regarding the application data component of the packet.

22.4 Operations

22.4.1 Announcing a Packet

The `rtems_multiprocessing_announce` directive is called by the MPCI layer to inform RTEMS that a packet has arrived from another node. This directive can be called from an interrupt service routine or from within a polling routine.

22.5 Directives

This section details the additional directives required to support RTEMS in a multiprocessor configuration. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

22.5.1 MULTIPROCESSING_ANNOUNCE - Announce the arrival of a packet

CALLING SEQUENCE:

```
void rtems_multiprocessing_announce( void );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive informs RTEMS that a multiprocessing communications packet has arrived from another node. This directive is called by the user-provided MPCI, and is only used in multiprocessor configurations.

NOTES:

This directive is typically called from an ISR.

This directive will almost certainly cause the calling task to be preempted.

This directive does not generate activity on remote nodes.

23 Directive Status Codes

`RTEMS_SUCCESSFUL` - successful completion

`RTEMS_TASK_EXITTED` - returned from a task

`RTEMS_MP_NOT_CONFIGURED` - multiprocessing not configured

`RTEMS_INVALID_NAME` - invalid object name

`RTEMS_INVALID_ID` - invalid object id

`RTEMS_TOO_MANY` - too many

`RTEMS_TIMEOUT` - timed out waiting

`RTEMS_OBJECT_WAS_DELETED` - object was deleted while waiting

`RTEMS_INVALID_SIZE` - invalid specified size

`RTEMS_INVALID_ADDRESS` - invalid address specified

`RTEMS_INVALID_NUMBER` - number was invalid

`RTEMS_NOT_DEFINED` - item not initialized

`RTEMS_RESOURCE_IN_USE` - resources outstanding

`RTEMS_UNSATISFIED` - request not satisfied

`RTEMS_INCORRECT_STATE` - task is in wrong state

`RTEMS_ALREADY_SUSPENDED` - task already in state

`RTEMS_ILLEGAL_ON_SELF` - illegal for calling task

`RTEMS_ILLEGAL_ON_REMOTE_OBJECT` - illegal for remote object

`RTEMS_CALLED_FROM_ISR` - invalid environment

`RTEMS_INVALID_PRIORITY` - invalid task priority

`RTEMS_INVALID_CLOCK` - invalid time buffer

`RTEMS_INVALID_NODE` - invalid node id

`RTEMS_NOT_CONFIGURED` - directive not configured

`RTEMS_NOT_OWNER_OF_RESOURCE` - not owner of resource

`RTEMS_NOT_IMPLEMENTED` - directive not implemented

`RTEMS_INTERNAL_ERROR` - RTEMS inconsistency detected

`RTEMS_NO_MEMORY` - could not get enough memory

24 Example Application

```
/*  example.c
 *
 *  This file contains an example of a simple RTEMS
 *  application.  It contains a Configuration Table, a
 *  user initialization task, and a simple task.
 *
 *  This example assumes that a board support package exists
 *  and invokes the initialize_executive() directive.
 */

#include "rtems.h"

rtems_task init_task();

#define INIT_NAME      rtems_build_name( 'A', 'B', 'C', ' ' ' )

rtems_initialization_tasks_table init_task = {
  { INIT_NAME,          /* init task name  "ABC" */
    1024,               /* init task stack size */
    1,                  /* init task priority */
    DEFAULT_ATTRIBUTES, /* init task attributes */
    init_task,          /* init task entry point */
    TIMESLICE,          /* init task initial mode */
    0                   /* init task argument */
  }
};

rtems_configuration_table User_Configuration_Table = {
  NULL,                 /* filled in by the BSP */
  65536,                /* executive RAM size */
  2,                    /* maximum tasks */
  0,                    /* maximum timers */
  0,                    /* maximum semaphores */
  0,                    /* maximum message queues */
  0,                    /* maximum messages */
  0,                    /* maximum partitions */
  0,                    /* maximum regions */
  0,                    /* maximum ports */
  0,                    /* maximum periods */
  0,                    /* maximum extensions */
  RTEMS_MILLISECONDS_TO_MICROSECONDS(10), /* number of ms in a tick */
  1,                    /* num of ticks in a timeslice  */
  1,                    /* number of user init tasks    */
  init_task_tbl,        /* user init task(s) table      */
  0,                    /* number of device drivers     */
  NULL,                 /* ptr to driver address table  */
  NULL,                 /* ptr to extension table */
  NULL                  /* ptr to MP config table */
};

task user_application(
  rtems_task_argument ignored
);

#define USER_APP_NAME  1  /* any 32-bit name; unique helps */

rtems_task init_task(
  rtems_task_argument ignored
)
{
  rtems_id tid;

  /* example assumes SUCCESSFUL return value */

  (void) rtems_task_create( USER_APP_NAME, 1, 1024,
                        RTEMS_NO_PREEMPT, RTEMS_FLOATING_POINT, &tid );
  (void) rtems_task_start( tid, user_application, 0 );
  (void) rtems_task_delete( SELF );
}



rtems_task user_application()

{
  /* application specific initialization goes here */

  while ( 1 )  {              /* infinite loop */

    /*  APPLICATION CODE GOES HERE
     *
     *  This code will typically include at least one
     *  directive which causes the calling task to
     *  give up the processor.
     */
  }
}
```

25 Glossary

active	A term used to describe an object which has been created by an application.
aperiodic task	A task which must execute only at irregular intervals and has only a soft deadline.
application	In this document, software which makes use of RTEMS.
ASR	see Asynchronous Signal Routine.
asynchronous	Not related in order or timing to other occurrences in the system.
Asynchronous Signal Routine	Similar to a hardware interrupt except that it is associated with a task and is run in the context of a task. The directives provided by the signal manager are used to service signals.
awakened	A term used to describe a task that has been unblocked and may be scheduled to the CPU.
big endian	A data representation scheme in which the bytes composing a numeric value are arranged such that the most significant byte is at the lowest address.
bit-mapped	A data encoding scheme in which each bit in a variable is used to represent something different. This makes for compact data representation.
block	A physically contiguous area of memory.
blocked	The task state entered by a task which has been previously started and cannot continue execution until the reason for waiting has been satisfied.
broadcast	To simultaneously send a message to a logical set of destinations.
BSP	see Board Support Package.
Board Support Package	A collection of device initialization and control routines specific to a particular type of board or collection of boards.
buffer	A fixed length block of memory allocated from a partition.
calling convention	The processor and compiler dependent rules which define the mechanism used to invoke subroutines in a high-level language. These rules define the passing of arguments, the call and return mechanism, and the register set which must be preserved.
Central Processing Unit	This term is equivalent to the terms processor and microprocessor.

chain	A data structure which allows for efficient dynamic addition and removal of elements. It differs from an array in that it is not limited to a predefined size.
coalesce	The process of merging adjacent holes into a single larger hole. Sometimes this process is referred to as garbage collection.
Configuration Table	A table which contains information used to tailor RTEMS for a particular application.
context	All of the processor registers and operating system data structures associated with a task.
context switch	Alternate term for task switch. Taking control of the processor from one task and transferring it to another task.
control block	A data structure used by the executive to define and control an object.
core	When used in this manual, this term refers to the internal executive utility functions. In the interest of application portability, the core of the executive should not be used directly by applications.
CPU	An acronym for Central Processing Unit.
critical section	A section of code which must be executed indivisibly.
CRT	An acronym for Cathode Ray Tube. Normally used in reference to the man-machine interface.
deadline	A fixed time limit by which a task must have completed a set of actions. Beyond this point, the results are of reduced value and may even be considered useless or harmful.
device	A peripheral used by the application that requires special operation software. See also device driver.
device driver	Control software for special peripheral devices used by the application.
directives	RTEMS' provided routines that provide support mechanisms for real-time applications.
dispatch	The act of loading a task's context onto the CPU and transferring control of the CPU to that task.
dormant	The state entered by a task after it is created and before it has been started.
Device Driver Table	A table which contains the entry points for each of the configured device drivers.
dual-ported	A term used to describe memory which can be accessed at two different addresses.

embedded	An application that is delivered as a hidden part of a larger system. For example, the software in a fuel-injection control system is an embedded application found in many late-model automobiles.
envelope	A buffer provided by the MPCI layer to RTEMS which is used to pass messages between nodes in a multiprocessor system. It typically contains routing information needed by the MPCI. The contents of an envelope are referred to as a packet.
entry point	The address at which a function or task begins to execute. In C, the entry point of a function is the function's name.
events	A method for task communication and synchronization. The directives provided by the event manager are used to service events.
exception	A synonym for interrupt.
executing	The task state entered by a task after it has been given control of the CPU.
executive	In this document, this term is used to referred to RTEMS. Commonly, an executive is a small real-time operating system used in embedded systems.
exported	An object known by all nodes in a multiprocessor system. An object created with the GLOBAL attribute will be exported.
external address	The address used to access dual-ported memory by all the nodes in a system which do not own the memory.
FIFO	An acronym for First In First Out.
First In First Out	A discipline for manipulating entries in a data structure.
floating point coprocessor	A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.
freed	A resource that has been released by the application to RTEMS.
global	An object that has been created with the GLOBAL attribute and exported to all nodes in a multiprocessor system.
handler	The equivalent of a manager, except that it is internal to RTEMS and forms part of the core. A handler is a collection of routines which provide a related set of functions. For example, there is a handler used by RTEMS to manage all objects.
hard real-time system	A real-time system in which a missed deadline causes the worked performed to have no value or to result in a catastrophic effect on the integrity of the system.

heap	A data structure used to dynamically allocate and deallocate variable sized blocks of memory.
heterogeneous	A multiprocessor computer system composed of dissimilar processors.
homogeneous	A multiprocessor computer system composed of a single type of processor.
ID	An RTEMS assigned identification tag used to access an active object.
IDLE task	A special low priority task which assumes control of the CPU when no other task is able to execute.
interface	A specification of the methodology used to connect multiple independent subsystems.
internal address	The address used to access dual-ported memory by the node which owns the memory.
interrupt	A hardware facility that causes the CPU to suspend execution, save its status, and transfer control to a specific location.
interrupt level	A mask used to by the CPU to determine which pending interrupts should be serviced. If a pending interrupt is below the current interrupt level, then the CPU does not recognize that interrupt.
Interrupt Service Routine	An ISR is invoked by the CPU to process a pending interrupt.
I/O	An acronym for Input/Output.
ISR	An acronym for Interrupt Service Routine.
kernel	In this document, this term is used as a synonym for executive.
list	A data structure which allows for dynamic addition and removal of entries. It is not statically limited to a particular size.
little endian	A data representation scheme in which the bytes composing a numeric value are arranged such that the least significant byte is at the lowest address.
local	An object which was created with the LOCAL attribute and is accessible only on the node it was created and resides upon. In a single processor configuration, all objects are local.
local operation	The manipulation of an object which resides on the same node as the calling task.
logical address	An address used by an application. In a system without memory management, logical addresses will equal physical addresses.
loosely-coupled	A multiprocessor configuration where shared memory is not used for communication.

major number	The index of a device driver in the Device Driver Table.
manager	A group of related RTEMS' directives which provide access and control over resources.
memory pool	Used interchangeably with heap.
message	A sixteen byte entity used to communicate between tasks. Messages are sent to message queues and stored in message buffers.
message buffer	A block of memory used to store messages.
message queue	An RTEMS object used to synchronize and communicate between tasks by transporting messages between sending and receiving tasks.
Message Queue Control Block	A data structure associated with each message queue used by RTEMS to manage that message queue.
minor number	A numeric value passed to a device driver, the exact usage of which is driver dependent.
mode	An entry in a task's control block that is used to determine if the task allows preemption, timeslicing, processing of signals, and the interrupt disable level used by the task.
MPCI	An acronym for Multiprocessor Communications Interface Layer.
multiprocessing	The simultaneous execution of two or more processes by a multiple processor computer system.
multiprocessor	A computer with multiple CPUs available for executing applications.
Multiprocessor Communications Interface Layer	A set of user-provided routines which enable the nodes in a multiprocessor system to communicate with one another.
Multiprocessor Configuration Table	The data structure defining the characteristics of the multiprocessor target system with which RTEMS will communicate.
multitasking	The alternation of execution amongst a group of processes on a single CPU. A scheduling algorithm is used to determine which process executes at which time.
mutual exclusion	A term used to describe the act of preventing other tasks from accessing a resource simultaneously.
nested	A term used to describe an ASR that occurs during another ASR or an ISR that occurs during another ISR.

node	A term used to reference a processor running RTEMS in a multiprocessor system.
non-existent	The state occupied by an uncreated or deleted task.
numeric coprocessor	A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.
object	In this document, this term is used to refer collectively to tasks, timers, message queues, partitions, regions, semaphores, ports, and rate monotonic periods. All RTEMS objects have IDs and user-assigned names.
object-oriented	A term used to describe systems with common mechanisms for utilizing a variety of entities. Object-oriented systems shield the application from implementation details.
operating system	The software which controls all the computer's resources and provides the base upon which application programs can be written.
overhead	The portion of the CPUs processing power consumed by the operating system.
packet	A buffer which contains the messages passed between nodes in a multiprocessor system. A packet is the contents of an envelope.
partition	An RTEMS object which is used to allocate and deallocate fixed size blocks of memory from an dynamically specified area of memory.
Partition Control Block	A data structure associated with each partition used by RTEMS to manage that partition.
pending	A term used to describe a task blocked waiting for an event, message, semaphore, or signal.
periodic task	A task which must execute at regular intervals and comply with a hard deadline.
physical address	The actual hardware address of a resource.
poll	A mechanism used to determine if an event has occurred by periodically checking for a particular status. Typical events include arrival of data, completion of an action, and errors.
pool	A collection from which resources are allocated.
portability	A term used to describe the ease with which software can be rehosted on another computer.
posting	The act of sending an event, message, semaphore, or signal to a task.

preempt	The act of forcing a task to relinquish the processor and dispatching to another task.
priority	A mechanism used to represent the relative importance of an element in a set of items. RTEMS uses priority to determine which task should execute.
priority inheritance	An algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task blocked waiting for that resource. This avoids the problem of priority inversion.
priority inversion	A form of indefinite postponement which occurs when a high priority tasks requests access to shared resource currently allocated to low priority task. The high priority task must block until the low priority task releases the resource.
processor utilization	The percentage of processor time used by a task or a set of tasks.
proxy	An RTEMS control structure used to represent, on a remote node, a task which must block as part of a remote operation.
Proxy Control Block	A data structure associated with each proxy used by RTEMS to manage that proxy.
PTCB	An acronym for Partition Control Block.
PXCB	An acronym for Proxy Control Block.
quantum	The application defined unit of time in which the processor is allocated.
queue	Alternate term for message queue.
QCB	An acronym for Message Queue Control Block.
ready	A task occupies this state when it is available to be given control of the CPU.
real-time	A term used to describe systems which are characterized by requiring deterministic response times to external stimuli. The external stimuli require that the response occur at a precise time or the response is incorrect.
reentrant	A term used to describe routines which do not modify themselves or global variables.
region	An RTEMS object which is used to allocate and deallocate variable size blocks of memory from a dynamically specified area of memory.
Region Control Block	A data structure associated with each region used by RTEMS to manage that region.
registers	Registers are locations physically located within a component, typically used for device control or general purpose storage.

remote	Any object that does not reside on the local node.
remote operation	The manipulation of an object which does not reside on the same node as the calling task.
return code	Also known as error code or return value.
resource	A hardware or software entity to which access must be controlled.
resume	Removing a task from the suspend state. If the task's state is ready following a call to the `rtems_task_resume` directive, then the task is available for scheduling.
return code	A value returned by RTEMS directives to indicate the completion status of the directive.
RNCB	An acronym for Region Control Block.
round-robin	A task scheduling discipline in which tasks of equal priority are executed in the order in which they are made ready.
RS-232	A standard for serial communications.
running	The state of a rate monotonic timer while it is being used to delineate a period. The timer exits this state by either expiring or being canceled.
schedule	The process of choosing which task should next enter the executing state.
schedulable	A set of tasks which can be guaranteed to meet their deadlines based upon a specific scheduling algorithm.
segments	Variable sized memory blocks allocated from a region.
semaphore	An RTEMS object which is used to synchronize tasks and provide mutually exclusive access to resources.
Semaphore Control Block	A data structure associated with each semaphore used by RTEMS to manage that semaphore.
shared memory	Memory which is accessible by multiple nodes in a multiprocessor system.
signal	An RTEMS provided mechanism to communicate asynchronously with a task. Upon reception of a signal, the ASR of the receiving task will be invoked.
signal set	A thirty-two bit entity which is used to represent a task's collection of pending signals and the signals sent to a task.
SMCB	An acronym for Semaphore Control Block.
soft real-time system	A real-time system in which a missed deadline does not compromise the integrity of the system.

sporadic task	A task which executes at irregular intervals and must comply with a hard deadline. A minimum period of time between successive iterations of the task can be guaranteed.
stack	A data structure that is managed using a Last In First Out (LIFO) discipline. Each task has a stack associated with it which is used to store return information and local variables.
status code	Also known as error code or return value.
suspend	A term used to describe a task that is not competing for the CPU because it has had a `rtems_task_suspend` directive.
synchronous	Related in order or timing to other occurrences in the system.
system call	In this document, this is used as an alternate term for directive.
target	The system on which the application will ultimately execute.
task	A logically complete thread of execution. The CPU is allocated among the ready tasks.
Task Control Block	A data structure associated with each task used by RTEMS to manage that task.
task switch	Alternate terminology for context switch. Taking control of the processor from one task and given to another.
TCB	An acronym for Task Control Block.
tick	The basic unit of time used by RTEMS. It is a user-configurable number of microseconds. The current tick expires when the `rtems_clock_tick` directive is invoked.
tightly-coupled	A multiprocessor configuration system which communicates via shared memory.
timeout	An argument provided to a number of directives which determines the maximum length of time an application task is willing to wait to acquire the resource if it is not immediately available.
timer	An RTEMS object used to invoke subprograms at a later time.
Timer Control Block	A data structure associated with each timer used by RTEMS to manage that timer.
timeslicing	A task scheduling discipline in which tasks of equal priority are executed for a specific period of time before being preempted by another task.
timeslice	The application defined unit of time in which the processor is allocated.
TMCB	An acronym for Timer Control Block.

transient overload	A temporary rise in system activity which may cause deadlines to be missed. Rate Monotonic Scheduling can be used to determine if all deadlines will be met under transient overload.
user extensions	Software routines provided by the application to enhance the functionality of RTEMS.
User Extension Table	A table which contains the entry points for each user extensions.
User Initialization Tasks Table	A table which contains the information needed to create and start each of the user initialization tasks.
user-provided	Alternate term for user-supplied. This term is used to designate any software routines which must be written by the application designer.
user-supplied	Alternate term for user-provided. This term is used to designate any software routines which must be written by the application designer.
vector	Memory pointers used by the processor to fetch the address of routines which will handle various exceptions and interrupts.
wait queue	The list of tasks blocked pending the release of a particular resource. Message queues, regions, and semaphores have a wait queue associated with them.
yield	When a task voluntarily releases control of the processor.

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.

Table of Contents