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 two parts. The most significant sixteen 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.

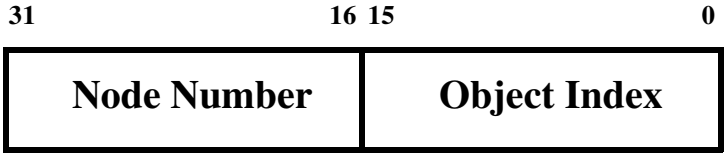

Figure 2-1 Object ID Composition

The two 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.