intel®

iRMX™ 86 NUCLEUS REFERENCE MANUAL

iRMX™ 86 OPERATING SYSTEM

Order Number: 9803122-04

iRMX™ 86 NUCLEUS REFERENCE MANUAL

Order Number: 9803122-04

REV.	REVISION HISTORY	PRINT DATE
-03	Describes high performance mailbox queues, 8087 NDP, cascaded interrupts, and enhanced interrupt processing; corrects various technical and typographical errors; and documents Release 3 of the iRMX 86 Operating System. Debugger and Terminal Handler information has been moved to separate manuals.	5/81
-04	Adds a chapter on configurable options, examples of system call usage, and all relevant information and system calls from iRMX 86 System Programmer's Reference Manual; corrects technical and typographical errors; and documents Release 5 of the iRMX 86 Operating System	9/82

BXP	Intel	iSBX	Multichannel
CREDIT	Int$_e$l	Library Manager	Multimodule
i	Intelevision	MCS	Plug-A-Bubble
ICE	Intellec	Megachassis	PROMPT
iCS	iOSP	Micromainframe	RMX/80
^{i}m	iRMX	Mircromap	System 2000
iMMX	iSBC	Multibus	UPI
Insite			

Printed in U.S.A./OM-001/5.2K/0283/AP

PREFACE

iRMX 86 provides an operating system for Intel iAPX 86-based microcomputers, including the iSBC 86/12A single board computer. It consists of a Nucleus, a Terminal Handler, a Debugger, a basic input/output system (BIOS), an extended input/output system (EIOS), an Application Loader, and a Human Interface. This manual describes the central portion of the Operating System, the Nucleus.

READER LEVEL

This manual is intended for both application and system programmers. It describes the basic features of the Nucleus and thoroughly documents the portion of the Nucleus that both application and system programmers require. It also contains detailed information about the features and system calls reserved for system programmers.

This manual is intended primarily as a source of Nucleus reference materials; it is only secondarily for instruction. If you are unfamiliar with the iRMX 86 Operating System, you should read the INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM prior to reading this manual.

CONVENTIONS

Throughout this manual, the following convention is used:

Reserved bits which should be set to zero.

Whenever this term is used, it means that the designated bits are not currently checked by the Nucleus. However, Intel reserves the right to establish meanings for these bits in future releases of the iRMX 86 Operating System. To ensure that your current system runs unchanged under future releases, you should set these bits to zero.

RELATED PUBLICATIONS

The following manuals provide additional information that may be helpful to readers of this manual.

- Introduction to the iRMX™ 86 Operating System, Order Number: 9803124
- iRMX™ 86 Installation Guide, Order Number: 9803125
- iRMX™ 86 Terminal Handler Reference Manual, Order Number: 143324
- iRMX™ 86 Debugger Reference Manual, Order Number: 143323
- iRMX™ 86 Basic I/O System Reference Manual, Order Number: 9803123
- iRMX™ 86 Extended I/O System Reference Manual, Order Number: 143308
- iRMX™ 86 Human Interface Reference Manual, Order Number: 9803202
- iRMX™ 86 Operator's Manual, Order Number: 144523
- iRMX™ 86 Loader Reference Manual, Order Number: 143381
- iRMX™ 86 Configuration Guide, Order Number: 9803126
- Guide to Writing Device Drivers for the iRMX™ 86 and iRMX 88™ I/O Systems, Order Number: 142926
- iRMX™ 86 Programming Techniques, Order Number: 142982
- iRMX™ 86 Pocket Reference, Order Number: 142861
- iSBC® 86/12A Hardware Reference Manual, Order Number: 9803074
- iOSP™ 86/88 Support Package Reference Manual, Order Number: 144437
- ISIS-II User's Guide, Order Number: 9800306
- PL/M-86 Programming Manual, Order Number: 9800466
- PL/M-86 Compiler Operating Introduction for 8080/8085 Based Development Systems, Order Number: 9800478
- The 8086 Family User's Manual, Order Number: 9800722

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

FIGURES

CHAPTER 1. OVERVIEW

The iRMX 86 Nucleus is the core of every iRMX 86 application system. Among the activities of the Nucleus are the following:

- Supplying scheduling functions
- Controlling access to system resources
- Providing for communication between individual processes
- Enabling the system to respond to external events

The Nucleus provides the building blocks from which the other subsystems (Basic I/O System, Extended I/O System, Application Loader, and Human Interface) and application systems are constructed. These building blocks are called objects and are classified into the following categories called object types:

- Tasks
- Jobs
- Segments
- Mailboxes
- Semaphores
- Regions
- Extension objects
- Composite objects

The following simplistic generalizations can be made regarding these types:

- Tasks are the active objects in a system. They do the work of the system.
- Jobs are the environments in which tasks do their work. An environment consists of tasks, the objects that tasks use, a directory where tasks can catalog objects so as to make them available to other tasks, and a pool of memory.
- Segments are pieces of memory, the medium that tasks use for communicating and for storing data.
- Mailboxes are the objects to which tasks go to send or receive other objects.

- Semaphores enable tasks to send signals to other tasks.
- Regions are objects that guard a specific collection of shared data.
- Extension objects are objects which designate new types of objects.
- Composite objects are objects of the new types designated by extension objects.

The Nucleus does extensive record-keeping of objects. It keeps track of each object by means of a 16-bit value called a token. The Nucleus provides a number of operators, called system calls, that tasks use to manipulate objects.

When using a system call, a task supplies parameter values, such as tokens, names, or other values, depending on the requirements of the system call. Some of the functions that tasks can perform with system calls are the following:

- Create objects
- Delete objects
- Send messages to other tasks
- Receive messages from other tasks
- Obtain information about objects
- Catalog objects with descriptive names
- Delete objects from catalogs

OBJECTS

Each of the eight object types discussed in this manual has unique characteristics. These characteristics are discussed in detail in the following sections.

TASKS

A task has two goals:

- Its primary goal is to do a specific piece of work.
- Its secondary goal is to obtain exclusive control of the processor so that it can progress toward its primary goal.

One of the main activities of the Nucleus is to arbitrate the competition that results when several tasks each want exclusive control over the processor. The Nucleus does this by maintaining, for each task, an execution state and a priority. The execution state for each task is, at any given time, either running, ready, asleep, suspended, or asleep-suspended. The priority for each task is an integer value between 0 and 255, inclusive, with 0 being the highest priority.

The arbitration algorithm that the Nucleus uses is that the running task is the ready task with the highest (numerically lowest) priority.

As viewed by the Nucleus, a task is merely a context consisting of values, some of which are the following:

- The task's priority
- The task's execution state
- A token for the job that contains the task

When a task becomes the running task, the following events occur, in order:

- The context of the previously running task is saved by the Nucleus
- The Nucleus sets the new running task's context
- The new task begins executing

The task continues to run until one of the following events occurs:

- The task removes itself from the ready state. For example, the task can suspend or delete itself; the task can attempt to receive a token for an object that has not yet been sent, in which case it might elect to wait (in the asleep state).
- The task (task A) is preempted when a higher priority task (task B) becomes ready. An example of how this could happen is that task B might previously have gone into the asleep state for a specific period of time. When the time period has passed, task B becomes ready again. Because it is then the highest priority ready task, task B becomes the running task.

JOBS

A <u>job</u> consists of tasks and the resources they need.

The jobs in a system form a family tree, with each job, except the root job, obtaining its resources from its parent. The tasks in the user jobs can create additional objects. If they create additional jobs, this enlarges the job tree.

The job tree, right after the initializaton of a system, is shown in Figure 1-1.

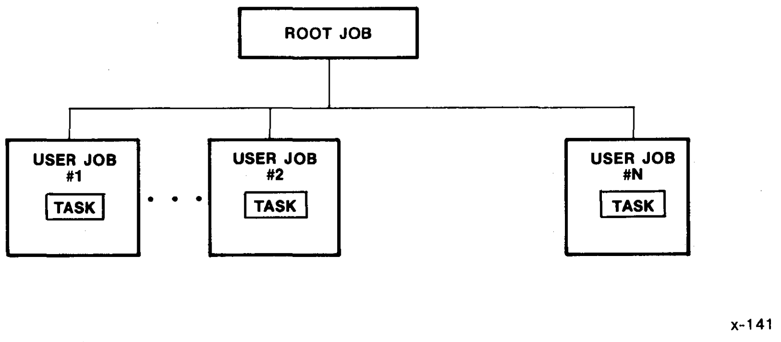


Figure 1-1. Initial Job Tree

Associated with each job is an object directory. Objects are known to the Nucleus by their respective tokens, but often, in the code that is executed by tasks, the objects are known by symbolic names. The object directory for a job is a place in memory where a task can catalog an object under a name. Other tasks that know the name can then use the directory to access the object.

Also associated with each job is a memory pool. This is an amount of memory which is allocated to the job and its descendents. All memory needed to create objects in the job comes from the memory pool.

SEGMENTS

A fundamental resource that tasks need is memory. Memory is allocated to tasks in the form of segments. A task needing memory requests a segment of whatever size it requires. The Nucleus attempts to create a segment from the memory pool given to the task's job when the job was created.

If there is not enough memory available, the Nucleus will try to borrow the needed memory from ancestors of the job. In this respect, the tree-structured hierarchy of jobs is instrumental in resource distribution.

MAILBOXES

A mailbox is one of three types of objects that can be used for intertask communication. When task A wants to send an object to task B, task A must send a token for the object to a mailbox, and task B must visit that mailbox, where, if a token for an object isn't there, it has the option of waiting for any desired length of time. When task B obtains the token, it can access the object. Sending a token for an object in this manner can achieve various purposes. The object might be a segment that contains data needed by the waiting task. On the other hand, the segment might be blank, and sending its token might constitute a signal to the waiting task. Another reason to send a token for an object might be to point out the object to the receiving task.

SEMAPHORES

A semaphore is a custodian of abstract "units". It dispenses units to tasks that request them, and it accepts units from tasks.

An example of typical semaphore use is mutual exclusion. Suppose your application system contains one I/O device which is being used for output by multiple tasks. To ensure that only one of these tasks can use the device at a given time, you can establish a semaphore which has one unit and require that tasks obtain the unit before using the device. A task wanting to use the device would request the unit from the semaphore. When it gets the unit, it can use the device and then return the unit to the semaphore. Because the semaphore has no units while the task is using the device, other tasks are effectively excluded from using the device.

REGIONS

A region is an iRMX 86 object that tasks can use to guard a specific collection of shared data. Once a task gains access to shared data through a region, the task can not be suspended or deleted by other tasks until it surrenders access. When the task currently using the shared data no longer needs access, it notifies the Operating System, which then allows the next task to access the shared data.

EXTENSION AND COMPOSITE OBJECTS

Whenever more than one job in your application system requires a function not supplied by the iRMX 86 Operating System, you can judiciously add new types of objects to your system to provide the needed function. The procedures that you must add to the Operating System in order to support the added function are called Operating System extensions. A type manager is an Operating System extension that can create new objects. A given type manager can only create one type of object, but can create numerous objects (called composite objects) of that object type. The object type is designated by an object called an extension object.

HANDLERS

Two kinds of events can be handled specially: exceptional conditions and interrupts. The remainder of this chapter describes the handlers for these events.

EXCEPTION HANDLERS

Tasks occasionally make errors. If an error occurs during an iRMX 86 system call, it causes an exceptional condition. The occurrence of an exceptional condition can, if desired, cause a transfer of control to the exception handler associated with the current task. The exception handler is a procedure that typically deals with the problem by one of the following methods:

- Correcting the cause of the problem and trying again
- Merely logging the error
- Deleting or suspending the task that caused the error

In regard to exception handlers, the designer of an iRMX 86-based system has two kinds of decisions to make for each task. The first decision concerns the choice of exception handlers. The task can have its own custom exception handler, it can use the exception handler for the job to which it belongs, or it can use the Intel-provided System Exception Handler. The second decision concerns when control goes to an exception handler. The task can direct control to the exception handler in avoidable (programmer) and/or unavoidable (environmental) conditions. If control is not directed to an exception handler, the responsibility for handling the exception falls upon the task.

INTERRUPT HANDLERS

To function effectively as a real-time system, an iRMX 86 application system must be responsive to external events. An interrupt handler, which is required for each source of external events (interrupts), is a procedure that is invoked by hardware or software for the purpose of responding to an asynchronous event. The handler takes control immediately and services the interrupt. When the interrupt handler is finished servicing the interrupt, it surrenders the processor, which returns to the interrupted procedure.

As part of its servicing, the interrupt handler can invoke a task to further process the interrupt. An interrupt handler invokes an interrupt task if the processing of an interrupt requires large amounts of time or if the processing requires those Nucleus system calls that interrupt handlers are prohibited from using.

CHAPTER 2. JOB MANAGEMENT

A job is an environment in which iRMX 86 objects such as tasks, mailboxes, semaphores, segments, and (offspring) jobs reside. In addition, a job has an object directory and a pool of memory. The job's memory pool provides the raw material from which objects can be created by the tasks in the job. Figure 2-1 illustrates some of the possible elements of a job.

Applications consist of one or more jobs. Jobs are independent but they may share resources. Each job has its own tasks and may have its own object directory. Objects may be shared between jobs, although each object is contained in only one job.

The programmer must decide whether tasks belong in the same job. In general, you should place tasks in the same job if:

- They have similar or related purposes
- They share many resources
- They have similar lifespans

JOB TREE AND RESOURCE SHARING

The jobs in a system are arranged in the form of a tree. The root is a job that is provided by the Nucleus. The remaining jobs, including jobs that are created dynamically while the system runs, are descendents of the root job. A job containing tasks that create other jobs is a parent job. A newly created job is a child of the job whose task created it.

Associated with each job is a set of limits. The limits of a job are as follows:

- Maximum allowable size of its object directory
- Maximum and minimum allowable sizes of its memory pool
- Maximum allowable number of simultaneously existing objects that it can contain
- Maximum allowable number of simultaneously existing tasks that it can contain
- Highest allowable priority of any task contained in it

You must specify these limits whenever you create a job. These limits, with the exception of object directory size, apply collectively to the job and all of its descendent jobs.

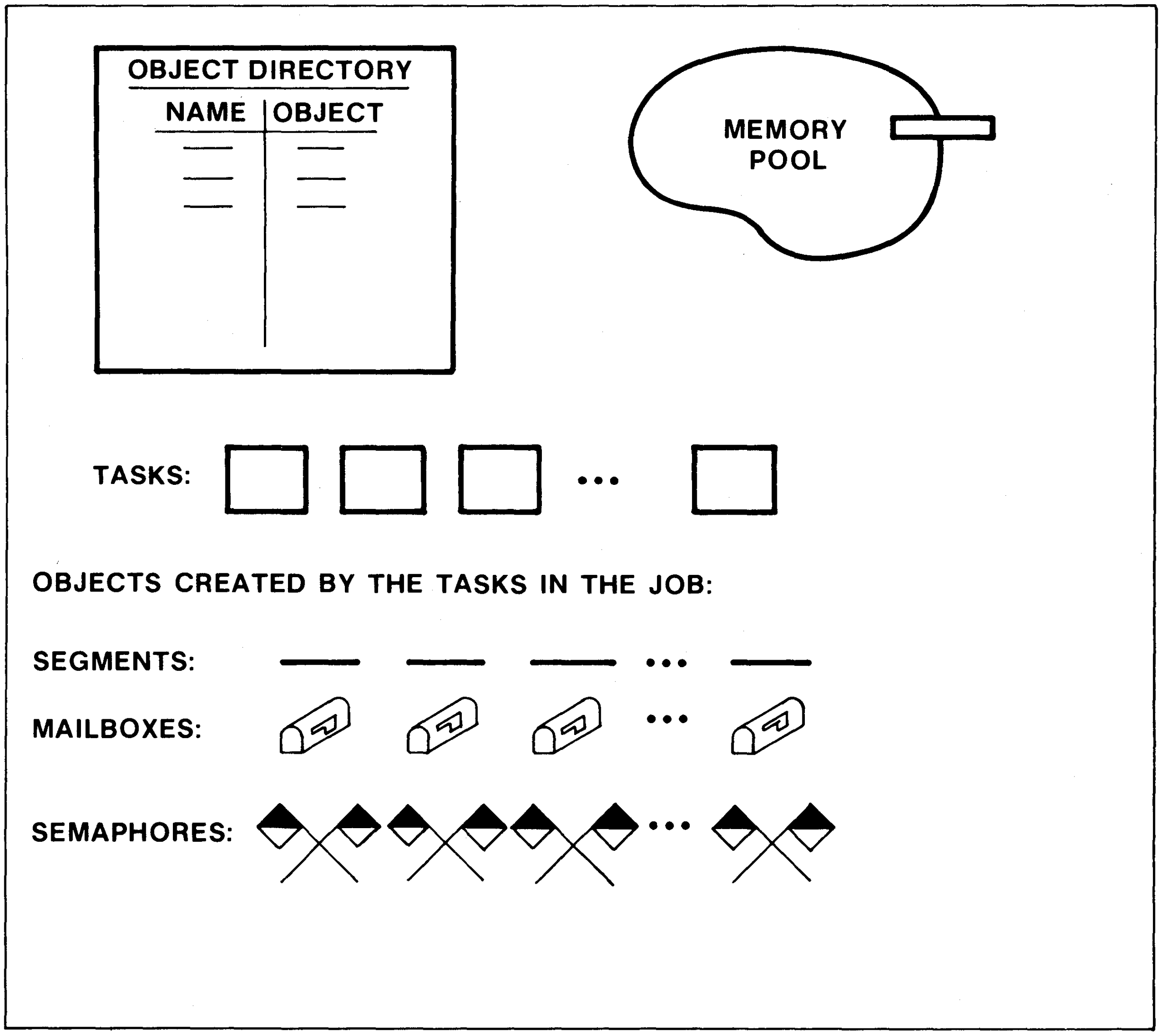


Figure 2-1. A Job

For example, suppose job A creates job B. When this happens:

- Sufficient memory to meet job B's minimum memory pool requirements is transferred from job A's memory pool to that of job B.
- The memory for job B including its object directory is taken from job A's memory pool.
- The numbers of tasks and total objects that job A can contain are reduced by the corresponding values specified for job B.
- The specified maximum priority for tasks in job B cannot exceed the maximum priority for tasks in job A.

If job B is later deleted, its resources are returned to job A.

JOB CREATION

A job is created with one task. The functions of this task should include doing some initializing for the new job. Initializing activities can include housekeeping and creating other objects in the new job.

When a task creates a job, it has the option of passing a token for a parameter object to the newly created job. The parameter object can be of any type and it can be used for any purpose. For example, the parameter object might be a segment containing data, arranged in a predefined format, needed by tasks in the new job. Tasks in the new job can obtain a token for the job's parameter object by means of the GET$TASK$TOKENS system call, described in Chapter 12.

JOB DELETION

Before a job can be deleted, all of its extension objects (see Chapter 10) and descendent jobs must be deleted. By using the OFFSPRING system call, the deleting task can probe down the job tree and find all of the descendents. Then it can delete them, beginning with descendents that have no children and working up the tree. After all of the descendents have been deleted, the task can delete the target job.

SYSTEM CALLS FOR JOBS

The following system calls manipulate jobs:

- CREATE$JOB --- creates a job with a task and returns a token for the job; resources for the new job are drawn from the resources of the job to which the invoking task belongs.
- DELETE$JOB --- deletes a childless job that contains no extension objects and returns the job's resources to its parent.
- OFFSPRING --- provides a segment containing tokens of the child jobs of the specified job.

CHAPTER 3. TASK MANAGEMENT

Tasks are the active objects in an iRMX 86 system. Each task is part of a job and is restricted to the resources that its job provides. Tasks should be written as PL/M-86 procedures, not as main modules.

The iRMX 86 Nucleus maintains a set of attributes for each task. Among these attributes are the priority and execution state of the task.

PRIORITY

A task's priority is an integer value between 0 and 255, inclusive. The lower the priority number, the higher the priority of the task. A high priority task has favored status as it competes with other tasks for the microprocessor.

Unless a task is involved in processing interrupts (see Chapter 8), its priority should be between 129 and 255. When a task having a priority in the range 0 to 128 is running, certain external interrupt levels are disabled, depending on the priority.

Also, if a task's code includes instructions that execute on the 8087 NPX (Numeric Processor Extension), that task should not have a priority high enough to disable the interrupt level of the NPX or a deadlock situation will result. The interrupt level of the 8087 NPX is configurable; refer to the iRMX 86 CONFIGURATION GUIDE for further information. Refer to Chapter 8 of this manual for a correlation between priorities and interrupt levels.

TASK STATES

A task is always in one of five execution states. The states are asleep, suspended, asleep-suspended, ready, and running.

THE ASLEEP STATE

A task is in the asleep state when it is waiting for a request to be granted. Also, a task can put itself to sleep for a specified amount of time by using the SLEEP system call.

THE SUSPENDED STATE

A task enters the suspended state when it is placed there by another task, when it is waiting for an interrupt, or when it suspends itself. Associated with each task is a suspension depth, which reflects the number of "suspends" outstanding against it. Each suspend operation must be countered with a resume operation before the task can leave the suspended state.

THE ASLEEP-SUSPENDED STATE

When a sleeping task is suspended, it enters the asleep-suspended state. In effect, it is then in both the asleep and suspended states. While asleep-suspended, the task's sleeping time might expire, putting it in the suspended state. Also, if another task resumes an asleep-suspended task, the latter task will enter the asleep state.

THE READY AND RUNNING STATES

A task is ready if it is not asleep, suspended, or asleep-suspended. For a task to become the running (executing) task, it must be the highest priority task in the ready state.

TASK STATE TRANSITIONS

The Nucleus does not allocate the processor to tasks in a time-slicing manner. Instead, as an iRMX 86 application system runs, events occur which cause tasks to pass from state to state. The iRMX 86 Operating System is, therefore, event-driven. Figure 3-1 shows the paths of transition between states.

The following list describes, by number, the events that cause the transitions in Figure 3-1. In the list, the migrating task is called "the task":

(1) When the task is created, it is placed in the ready state .

(2) The task goes from the ready state to the running state when one of the following occurs:

- The task has just become ready and has higher priority than does any other ready task.
- The task is ready, no other ready task has higher priority, no other task of equal priority has been ready for a longer time, and the previously running task has just left the running state by (4), (6), or (10).

(3) The task goes from the running state to the ready state when the task is preempted by a higher priority task that has just become ready.

(4) The task goes from the running state to the asleep state when one of the following occurs:

- the task puts itself to sleep (by the SLEEP system call.)

- The task makes a request (by the RECEIVE$MESSAGE, RECEIVE$UNITS, RECEIVE$CONTROL, or LOOKUP$OBJECT system call) that cannot be granted immediately and expresses, in the request, its willingness to wait.

(5) The task goes from the asleep state to the ready state or from the asleep-suspended state to the suspended state when one of the following occurs:

- The time period specified in the invocation of the SLEEP system call expires.

- The task's designated waiting period expires without its request being granted.

- The task's request is granted (because another task called either the SEND$MESSAGE, SEND$UNITS, SEND$CONTROL, or CATALOG$OBJECT system call; these calls correspond to those mentioned in (4), above).

(6) The task goes from the running state to the suspended state when the task suspends itself (by the SUSPEND$TASK or WAIT$INTERRUPT system call).

(7) The task goes from the ready state to the suspended state or from the asleep state to the asleep-suspended when the task is suspended by another task (by the SUSPEND$TASK system call).

(8) The task remains in the suspended state or the asleep-suspended state when one of the following occurs:

- (same as (7)) or

- The task has a suspension depth greater than one and the task is resumed by another task (by the RESUME$TASK system call).

(9) The task goes from the suspended state to the ready state or from the asleep-suspended state to the asleep state when the task has a suspension depth of one and the task is resumed by another task (by the RESUME$TASK or SIGNAL$INTERRUPT system call) or when a task awaiting an interrupt receives the interrupt.

(10) The task goes from any state to non-existence when it is deleted (by the DELETE$TASK, DELETE$JOB, or RESET$INTERRUPT system call).

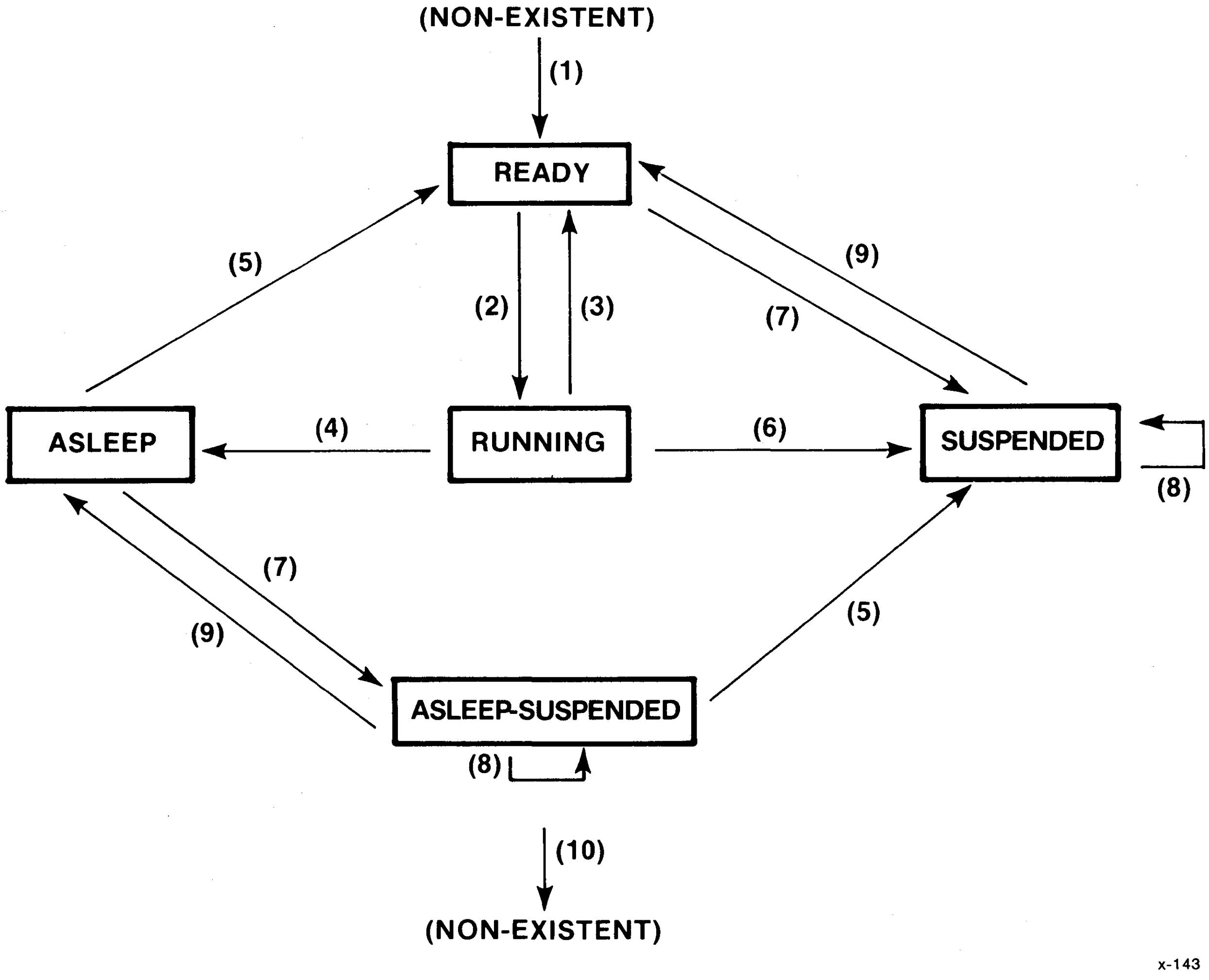


Figure 3-1. Task State Transition Diagram

ADDITIONAL TASK ATTRIBUTES

In addition to priority, execution state, and suspension depth, the Nucleus maintains current values of the following attributes for each existing task: containing job, its PL/M-86 register context, starting address of its exception handler (see Chapter 7), its exception mode (see Chapter 7), whether or not it is an interrupt task (see Chapter 8) and whether the task uses the 8087 NPX.

TASK RESOURCES

When a task is created, the Nucleus takes any resources that it needs at that time (such as memory for a stack) from the task's containing job. If the task is subsequently deleted, those resources are returned to the job.

SYSTEM CALLS FOR TASKS

The following system calls are provided for task manipulation:

- CREATE$TASK --- creates a task and returns a token for it.
- DELETE$TASK --- deletes a task from the system.
- SUSPEND$TASK --- increases a task's suspension depth by one; suspends the task if it is not already suspended.
- RESUME$TASK --- decreases a task's suspension depth by one; if the depth becomes zero and the task was suspended, it then becomes ready; if the depth becomes zero and the task was asleep-suspended, then it goes into the asleep state.
- SLEEP --- places the calling task in the asleep state for a specified amount of time.
- GET$TASK$TOKENS --- returns to the calling task a token for either itself, its job, its job's parameter object, or the root job, depending on which option is specified in the call.
- GET$PRIORITY --- returns the priority of the specified task.

CHAPTER 4. EXCHANGE MANAGEMENT

The iRMX 86 Nucleus provides exchanges to facilitate intertask communication, synchronization, and mutual exclusion. When a task uses an exchange, it is always acting either as a sender or as a receiver. There are three kinds of exchanges: mailboxes, semaphores, and regions. If the exchange is a mailbox, one task will send a token for an object to the mailbox; another task will go to the mailbox to receive the object's token. If the exchange is a semaphore, either a task is receiving units from the semaphore, or it is sending units to the semaphore. Regions are discussed in Chapter 9 of this manual.

MAILBOXES

The principal function of mailboxes is to support intertask communication. A sending task uses a mailbox to pass the token for an object to another task. For example, the object might be that of a segment containing data needed by the receiving task.

NOTE

This chapter refers to the passing of objects between jobs or between tasks. In fact, tokens rather than objects are passed.

MAILBOX QUEUES

Each mailbox has two queues, one for tasks that are waiting to receive objects, the other for objects that have been sent by tasks but have not yet been received. The Nucleus sees that waiting tasks receive objects as soon as they are available, so, at any given time, at least one of the mailbox's queues is empty.

MAILBOX MECHANICS

When a task sends a token to a mailbox, using the SEND$MESSAGE system call, one of two things happens. If no tasks are waiting at the mailbox, the object is placed at the rear of the object queue (which might be empty). Object queues are processed in a first-in/first-out (FIFO) manner, so the object remains in the queue until it makes its way to the front and is given to a task.

If, on the other hand, there are tasks waiting, the receiving task, which has been asleep, goes either from the asleep state to the ready state or from the asleep-suspended state to the suspended state.

NOTE

If the receiving task has a higher priority than the sending task, then the receiving task preempts the sender and becomes the running task.

When a task attempts to receive an object from a mailbox via the RECEIVE$MESSAGE system call, and the object queue at the mailbox is not empty, the task receives the object immediately and remains ready. However, if there are no objects at the mailbox there are two possibilities:

- If the task, in its request, elects to wait, it is placed in the mailbox's task queue and is put to sleep. If the designated waiting period elapses before the task gets an object, the task is made ready and receives an E$TIME exceptional condition (see Chapter 7).
- If the task is not willing to wait, it remains ready and receives an E$TIME exceptional condition.

A task has the option, when using the SEND$MESSAGE system call, of specifying that it wants acknowledgment from the receiving task. Thus, any task using the RECEIVE$MESSAGE system call should check to see if an acknowledgment has been requested. For details, see the description of the RECEIVE$MESSAGE system call in Chapter 12.

As stated earlier, the object queue for a mailbox is processed in a first-in/first-out manner. However, the task queue of a mailbox can be either first-in/first-out or priority-based, with higher-priority tasks toward the front of the queue. When a task creates a mailbox, the task specifies which kind of task queue the mailbox is to have.

HIGH PERFORMANCE OBJECT QUEUE

Directly associated with each mailbox is a high performance object queue. A task, when creating a mailbox with CREATE$MAILBOX, can specify the number of objects this queue can hold, from 4 to 60. By using this high performance object queue, the task can greatly improve the performance of SEND$MESSAGE and RECEIVE$MESSAGE when these calls actually get or place objects on the queue (it has no effect when tasks are already waiting at the task queue). When more objects than the high performance queue can hold are queued at a mailbox, the objects overflow into a slower queue whose size is limited only by the amount of memory in the job containing the mailbox.

The high performance queue obtains its high speed because the Nucleus allocates memory space for it as soon as the mailbox is created. This memory space is permanently allocated to the mailbox, even if no objects are queued there. No space is allocated for the overflow portion of the queue until the space is needed to contain objects. Thus the overflow portion of the queue is slower.

The user must weigh performance against size when deciding how large to make the high performance queue. Specifying a high performance queue that is too large results in a waste of memory. Conversely, a smaller queue that is constantly overflowing does not realize all possible performance benefits. Appendix C lists the memory usage algorithm for high performance queues.

SYSTEM CALLS FOR MAILBOXES

The following system calls manipulate mailboxes:

- CREATE$MAILBOX --- creates a mailbox and returns a token for it.
- DELETE$MAILBOX --- deletes a mailbox from the system.
- SEND$MESSAGE --- sends an object to a mailbox.
- RECEIVE$MESSAGE --- sends the calling task to a mailbox for an object; the task has the option of waiting if no objects are present.

SEMAPHORES

A semaphore is a custodian of abstract units. A task uses a semaphore either by requesting a specific number of units from it via the RECEIVE$UNITS system call or by releasing a specific number of units to it via the SEND$UNITS system call. Although these operations do not support communication of data, they facilitate mutual exclusion, synchronization, and resource allocation.

SEMAPHORE QUEUE

Semaphores have only one queue - a task queue. As is the case with mailboxes, the task queue is either first-in/first-out or priority-based. The queueing scheme to be used is specified for each semaphore at the time of its creation.

SEMAPHORE MECHANICS

A semaphore might simultaneously have both tasks in its queue and units in its custody. The allocation scheme used by semaphores is the reason for this. That scheme is best understood by imagining that the semaphore is trying, at all times, to satisfy the request of the task which is at the front of the semaphore's task queue. Only when it can provide as many units as the task requested does it award units, and then it does so immediately.

When a task uses the CREATE$SEMAPHORE system call, it must supply two values. One value specifies the initial number of units to be in the new semaphore's custody. The other value sets an upper limit on the number of units that the semaphore is allowed to keep at any given time. The lower limit is automatically zero.

When a task requests units from a semaphore via the RECEIVE$UNITS system call, the request must be within the specified maximum for that semaphore; otherwise, the request is invalid and causes an E$LIMIT exceptional condition. If a task's request for units is valid and both

- the size of the request is within the semaphore's current supply of units and
- the task is - or would be if queued - at the front of the semaphore's task queue,

then the request is granted immediately and the task remains ready. Otherwise, one of the following applies:

- The task, in its request, elects to wait. It is placed in the semaphore's task queue and is put to sleep. If the designated waiting period elapses before the task gets its requested units, the task is made ready and receives an E$TIME exceptional condition.
- The task is not willing to wait. It remains ready and receives an E$TIME exceptional condition.

Suppose, for example, that two tasks, A and B, are waiting at a semaphore, with A at the front of the queue. The semaphore has no units, A wants 3 units, and B wants 1 unit. The following three separate cases illustrate the mechanics of the semaphore:

- If the semaphore is sent 2 units, both A and B remain asleep in the semaphore's queue. Note that B's modest request is not satisfied because A is ahead of B in the queue.
- If, instead, the semaphore is sent 3 units, A receives the units and awakens, while B remains asleep in the queue.
- If, instead, the semaphore is sent 4 units, A and B both receive their requested units and are awakened. A is awakened first.

When a task sends units to a semaphore, the task remains ready. Sending units to a semaphore causes an E$LIMIT exceptional condition if it pushes the semaphore's supply above the designated maximum. The number of units in the custody of the semaphore remains unchanged.

NOTE

It is possible that a task sending units to a semaphore can be preempted by a higher priority task becoming ready as a result of getting its requested units.

SYSTEM CALLS FOR SEMAPHORES

The following system calls manipulate semaphores:

- CREATE$SEMAPHORE --- creates a semaphore and returns a token for it.
- DELETE$SEMAPHORE --- deletes a semaphore from the system.
- SEND$UNITS --- adds a specific number of units to the supply of a semaphore.
- RECEIVE$UNITS --- asks for a specific number of units from a semaphore.

CHAPTER 5. MEMORY MANAGEMENT

Occasionally a task needs additional memory. By using Nucleus system calls for allocating and deallocating memory, tasks can usually satisfy their memory needs.

SEGMENTS

Allocated memory is treated as a collection of segments. A segment is a contiguous sequence of 16-byte paragraphs, with its starting (base) address evenly divisible by 16. The base address functions as the token for the segment. The Nucleus maintains, as attributes, the base address, the length in bytes of each segment, and a token for the containing job.

When a task needs a segment, it can request one of the desired length via the CREATE$SEGMENT system call. If enough memory is available, the Nucleus returns a token for the segment.

NOTE

The token for a segment can be used as the base portion of a pointer to the segment. Thus, the token can be used as a base address (as when writing a message in the segment) or as an object reference (as when sending the segment-with-message to a mailbox). The PL/M-86 SELECTOR data type is especially useful in referring to the segment.

MEMORY POOLS

A memory pool is the memory available to a job and its descendents. Each job has a memory pool. When a job is created, the memory for its pool is allocated from the pool of its parent job. Thus, there is effectively a tree-structured hierarchy of memory pools, identical in structure to the hierarchy of jobs. Memory that a job borrows from its parent remains in the pool of the parent as well as being in the pool of the child. Such memory, however, is available for use only by tasks in the child job, and not by tasks in the parent job. Figure 5-1 illustrates the relationship between the job and memory hierarchies. In the figure, the pool sizes shown are actually the maximum sizes of those pools.

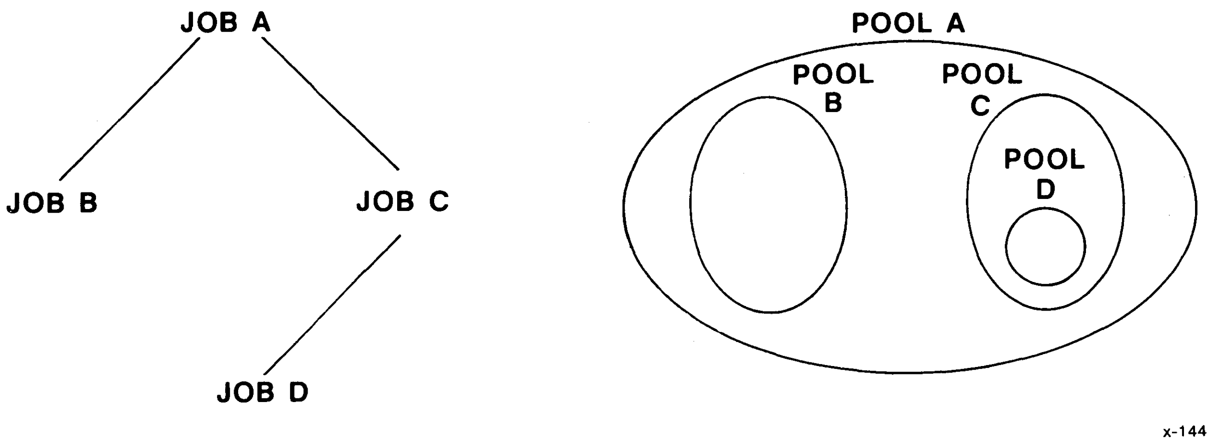


Figure 5-1. Comparison of Job and Memory Hierarchies

CONTROLLING POOL SIZE

Two parameters, pool$min and pool$max, of the CREATE$JOB system call, dictate the range of sizes (in 16-byte paragraphs) of a new job's memory pool. Initially, the pool size is equal to pool$min, the pool minimum. Memory allocated to tasks in the job is still considered to be in the job's pool. A task needing to know about its job's pool may use the GET$POOL$ATTRIB system call to obtain pool$min, pool$max, the initial pool size, the number of paragraphs currently available, and the number of paragraphs currently allocated.

A task may alter the pool minimum attribute for its job by means of the SET$POOL$MIN system call; pool$min must lie in the range from 0 to pool$max, the pool maximum. If a subsequent call to SET$POOL$MIN increases the pool's minimum size, and the current pool size is less than the new minimum, no memory is borrowed immediately from the parent job. Rather, memory is automatically borrowed as it is requested by tasks in the job, until the new minimum is reached. At that time, the new value of the pool minimum attribute becomes a lower bound for the job's pool size.

MOVEMENT OF MEMORY BETWEEN JOBS

When a task tries to create a segment (or an object of any other type), and the unallocated part of its job's pool is not sufficient to satisfy the request, the Nucleus tries to borrow more memory from the job's parent (and then, if necessary, from its parent's parent, and so on). Such borrowing increases the pool size of the borrowing job and is thus restricted by the pool maximum attribute of the borrowing job.

When a job is deleted, the memory in its pool becomes unallocated, and access to it is given back to the parent job. The smallest contiguous piece of memory that a job may borrow from its parent is a configuration parameter. The subject of configuration is covered in the iRMX 86 CONFIGURATION GUIDE.

Observe that, if a job has equal pool minimum and pool maximum attributes, then its pool is fixed at that common value. This means that, once it has this amount, the job may not borrow memory from its parent.

MEMORY ALLOCATION

The memory pool of a job consists of two classes of memory: allocated and unallocated. Memory in a job is unallocated unless it has been requested, either explicitly or implicitly, by tasks in the job or unless it is on loan to a child job. A task's request for memory is explicit when it calls the CREATE$SEGMENT system call. A request is implicit when the task attempts to create any type of object other than a segment.

The Nucleus borrows small amounts of memory from a job's pool each time a task in that job creates an object. This memory is needed for bookkeeping purposes. When the object is deleted, the borrowed memory is returned to the pool. Appendix C lists these memory requirements.

When a task no longer needs a segment, it can return the segment to the unallocated part of the job's pool by using the DELETE$SEGMENT system call. Figure 5-2 shows how memory "moves."

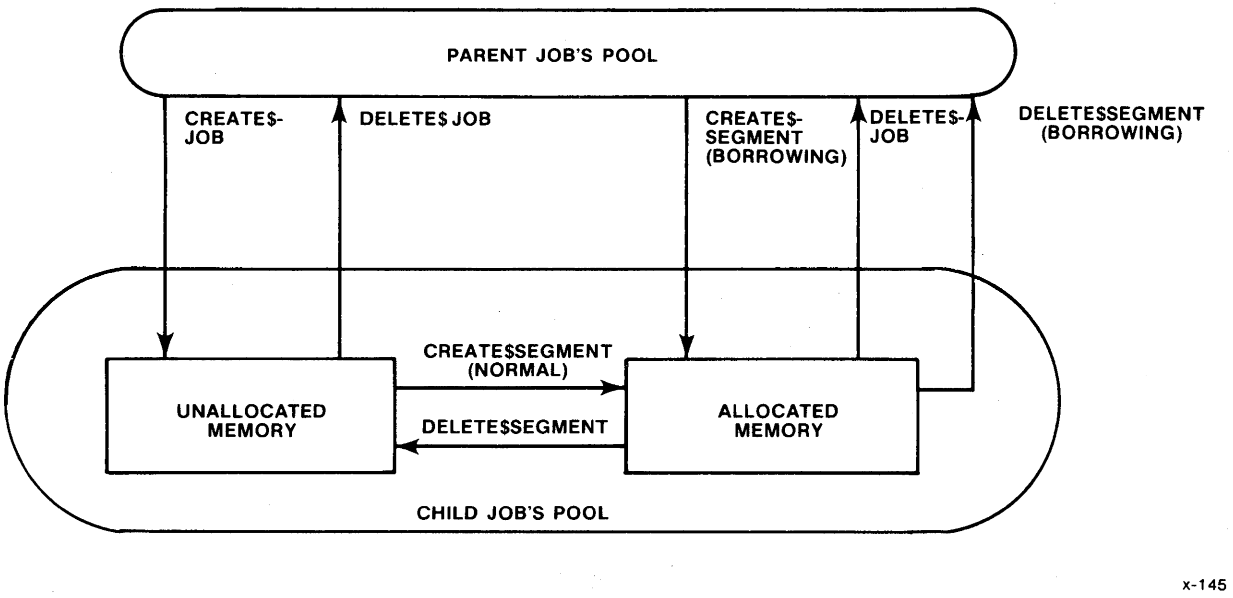


Figure 5-2. Memory Movement Diagram

SYSTEM CALLS FOR SEGMENTS

The following system calls manipulate segments:

- CREATE$SEGMENT --- creates a segment and returns a token for it.
- DELETE$SEGMENT --- returns a segment to the pool from which it was allocated.
- GET$SIZE --- returns the size, in bytes, of a segment.
- SET$POOL$MIN --- enables a task to change the pool minimum attribute of its job's pool.
- GET$POOL$ATTRIB --- returns the following memory pool attributes of the calling task's job: pool minimum, pool maximum, initial size, number of allocated paragraphs, and number of available paragraphs.

CHAPTER 6. OBJECT MANAGEMENT

A few iRMX 86 Nucleus system calls apply to all objects. These system calls allow tasks to inquire about an object's type and to use object directories.

INQUIRING ABOUT OBJECT TYPES

The GET$TYPE system call enables a task to present a token to the Nucleus and get an object's type code in return. (Type codes for Nucleus objects are listed in Appendix B.) This is useful, for example, when a task is expecting to receive objects of several different types. With the object's type code, the task can use the appropriate system calls for the object.

USING OBJECT DIRECTORIES

Each job has its own object directory. An entry in an object directory consists of a token for an object and the object name. The name contains from one to twelve characters, where a character is a one-byte value (from 0 to 0FFH). Such a feature is often needed because some tasks might only know some objects by their associated names.

By using the LOOKUP$OBJECT system call, a task can present the name of an object to the Nucleus. The Nucleus consults the object directory corresponding to the specified job and, if the object has been cataloged there, returns the token.

NOTE

In object directories, upper and lower case alphabetic characters are treated as being different. The Nucleus sees the name as just a string of bytes. It does not interpret these bytes as ASCII characters.

If the object has not yet been cataloged, and the task is not willing to wait, the task remains ready and receives an E$TIME exceptional condition. However, if the task is willing to wait, it is put to sleep; there are two possibilities:

- If the designated waiting period elapses before the task gets its requested token, the task is made ready and receives an E$TIME exceptional condition (see Chapter 7).

- If the task gets its requested token within the designated waiting period, it is made ready with no exceptional condition. This case is possible because another task can, while the requesting task is waiting, catalog the appropriate entry in the specified object directory.

When a task wants to share an object with the other tasks in a job (not necessarily its own job), it can use the CATALOG$OBJECT system call to put the object in that job's object directory. Typically, this is done by the creator of the object. Likewise, entries can be removed from a directory by the UNCATALOG$OBJECT system call.

What is required, when using an object directory, is the token of the job whose directory is to be used. The root job's object directory, called the root object directory, is special in that its token is easily accessible. Any task can call the GET$TASK$TOKENS system call to obtain the token of the root job.

SYSTEM CALLS FOR ANY OBJECTS

The following system calls manipulate objects:

- CATALOG$OBJECT --- places an object in an object directory.
- UNCATALOG$OBJECT --- removes an object from an object directory.
- LOOKUP$OBJECT --- accepts a cataloged name of an object and returns a token for it.
- GET$TYPE --- accepts a token for an object and returns its type code.

CHAPTER 7. EXCEPTIONAL CONDITION MANAGEMENT

When a task invokes an iRMX 86 system call, the results are sometimes not what the task is trying to achieve. For example, suppose a task requests memory that is not available or uses an invalid token as a parameter. In such cases, the system must inform the task that an error occurred. Whenever a task makes a system call, the means of communicating the success or failure of the call is the condition code.

TYPES OF EXCEPTIONAL CONDITIONS

Table 7-1 is a list of Nucleus conditions and their codes. The conditions that represent failure are called exceptional and are classified as programmer errors or environmental conditions. An exceptional condition that is preventable by the calling task is a programmer error. In contrast, exceptional conditions due to environmental circumstances of which the task could have no awareness are considered environmental conditions.

Table 7-1 lists the possible conditions, with their associated numeric codes and mnemonics. Values not used as numeric codes are reserved.

EXCEPTION HANDLERS

The iRMX 86 Nucleus supports exception handlers. Their purpose is to deal with the errors that tasks encounter in making system calls. How an exception handler deals with an exceptional condition is a matter of programmer discretion. In general, a handler performs one of the following actions:

- Logs the error.
- Deletes or suspends the task that erred.
- Ignores the error. If this option is taken, the system continues as if no error had occurred. Continuing under such circumstances is generally unwise, however.

An exception handler is written as a procedure with four parameters passed in the following order:

- The condition code (WORD).
- A code (BYTE) indicating which parameter, if any, was faulty in the call (1 for first, 2 for second, etc., 0 if none).
- A reserved (WORD) parameter.
- A second reserved (WORD) parameter.

ASSIGNING AN EXCEPTION HANDLER

A task may use the SET$EXCEPTION$HANDLER system call to declare its own exception handler. Otherwise, the task inherits the exception handler of its job. A job can receive its own exception handler at the time of its creation. If it doesn't, the job inherits the system exception handler. Thus, the Nucleus can always find an exception handler for the running task.

A system exception handler is provided as part of the iRMX 86 Operating System. Depending on a configuration option, it either deletes or suspends any task on whose behalf it is invoked. The iRMX 86 CONFIGURATION GUIDE describes this configuration option.

Users wanting to write their own exception handlers should compile them under the PL/M-86 LARGE control.

Any task can have the Debugger as its exception handler; see the description in Chapter 12 of the SET$EXCEPTION$HANDLER system call for instructions on how to dynamically make such an assignment. Alternatively, the Debugger or any other routine can be made the system exception handler statically; see the iRMX 86 CONFIGURATION GUIDE for information on how to do this.

INVOKING AN EXCEPTION HANDLER

An exception handler normally receives control when an exceptional condition occurs. However, when a task encounters an exceptional condition, it need not always have control passed to its exception handler. The factor that determines whether control passes to the exception handler is the task's exception mode. This attribute has four possible values, each of which specifies the circumstances under which the exception handler is to get control in the event of an exceptional condition. These circumstances are:

- Programmer errors only.
- Environmental conditions only.
- All exceptional conditions.
- No exceptional conditions.

When the Nucleus detects that a task has caused an exceptional condition in making a system call, it compares the type of the condition with the calling task's exception mode. If a transfer of control is indicated, the Nucleus passes control to the exception handler on behalf of the task. The exception handler then deals with the problem, after which control returns to the task, unless the exception handler deleted the task. While the exception handler is executing, the errant task is still regarded by the Nucleus to be the running task.

When a task is created, its exception mode is set to its job's default exception mode. The task can change its exception handler and exception mode attributes by using the SET$EXCEPTION$HANDLER system call.

HANDLING EXCEPTIONS IN-LINE

If a task's exception mode attribute does not direct the Nucleus to transfer control to the task's exception handler, the responsibility for dealing with an error falls upon the task.

Each system call has as its last parameter a POINTER to a WORD. After a system call, the Nucleus returns the resulting condition code to this WORD. By checking this WORD after each system call, a task can ascertain whether the call was successful. (See Table 7-1 for condition codes.) If the call was not successful, the task can learn which exceptional condition it caused. This information can sometimes enable the task to recover. In other cases more information is needed.

If a system call returns an exception code to indicate an unsuccessful call, all other output parameters of that system call are undefined.

NOTE

If an exceptional condition is caused by an invalid parameter, an exception handler, which is passed the parameter number of the first invalid parameter, should handle the condition.

Table 7-1. Conditions and Their Codes

CATEGORY/ MNEMONIC	MEANING	NUMERIC CODE	
		HEX	DECIMAL
Normal			
E$OK	The most recent system call was successful.	0H	0
Exceptional			
Environmental Conditions			
E$TIME	A time limit (possibly a limit of zero time) expired without a task's request being satisfied.	1H	1
E$MEM	There is not sufficient memory avail-able to satisfy a task's request.	2H	2
E$LIMIT	A task attempted an operation which, if it had been successful, would have violated a Nucleus-enforced limit.	4H	4
E$CONTEXT	A system call was issued out of context. or the Nucleus was asked to perform an impossible operation.	5H	5
E$EXIST	A token parameter has a value which is not the token of an existing object.	6H	6
E$STATE	A task attempted an operation which would have caused an impossible transition of a task's state.	7H	7
E$NOT$CON-FIGURED	The system call being attempted is not not part of the present software configuration.	8H	8
E$INTER-RUPT$SAT-URATION	An interrupt task has accumulated the maximum allowable amount of SIGNAL$IN-TERRUPT requests.	9H	9
E$INTER-RUPT$OV-ERFLOW	An interrupt task has accumulated more than the maximum allowable amount of SIGNAL$INTERRUPT requests.	0AH	10

Table 7-1. Conditions and Their Codes (continued)

CATEGORY/ MNEMONIC	MEANING	NUMERIC CODE	
		HEX	DECIMAL
Programmer Errors			
E$ZERO$-DIVIDE	A task attempted to divide by zero.	8000H	32768
E$OVERFLOW	An overflow interrupt occurred.	8001H	32769
E$TYPE	A token parameter referred to an existing object that is not of the required type.	8002H	32770
E$PARAM	A parameter which is neither a token nor an offset has an illegal value.	8004H	32772
E$BAD$CALL	A task wrote over the interface library or attempted a restricted software interrupt.	8005H	32773
E$ARRAY$BOUNDS	Hardware or Language has detected an array overflow.	8006H	32774
E$NDP$ERROR	An 8087 Numeric Processor Extension error has been detected; Operating System extensions can return the status of the 8087 to the exception handler.	8007H	32775
E$CHECK$-EXCEPTION	Language has detected a software interrupt type 17.	8017H	32791

SYSTEM CALLS FOR EXCEPTION HANDLERS

The following system calls manipulate exception handlers:

- SET$EXCEPTION$HANDLER --- sets the exception handler and exception mode attributes of the calling task.
- GET$EXCEPTION$HANDLER --- returns to the calling task the current values of its exception handler and exception mode attributes.

CHAPTER 8. INTERRUPT MANAGEMENT

Interrupts and interrupt processing are central to real-time computing. External events occur asynchronously with respect to the internal workings of an iRMX 86 application system. An interrupt, signalling the occurrence of an external event, triggers an implicit "call" to a location specified in a section of memory known as the interrupt vector table. From there, control is redirected to an interrupt procedure called an interrupt handler. At this point, one of two things happens. If handling the interrupt takes little time and requires no system calls, other than certain interrupt-related system calls, the interrupt handler processes the interrupt. Otherwise, the interrupt handler invokes an interrupt task which deals with the interrupt. After the interrupt has been serviced, control returns to the ready application task with highest priority.

INTERRUPT MECHANISMS

There are three major concepts in interrupt processing: the interrupt vector table, interrupt levels, and disabling interrupt levels.

THE INTERRUPT VECTOR TABLE

The interrupt vector table is composed of 256 vectors. The vectors are numbered 0 to 255. A number of the interrupt vectors are reserved and therefore are not available to be defined by user tasks. The vectors are allocated as follows:

0	divide by zero
1	single step (used by the iSBC 957A/B package)
2	non-maskable interrupt (used by the iSBC 957A/B package)
3	one byte interrupt instruction (used by the iRMX 86 Debugger and the iSBC 957A/B package)
4	interrupt on overflow (used by the hardware)
5	runtime array bounds error (used by compilers and assembler)
6-55	reserved
56-63	reserved for external interrupts (PIC master levels)
64-127	reserved for external interrupts (PIC slave levels)
128-183	unused (available to users)
184-223	reserved
224-255	described in Chapter 10

INTERRUPT LEVELS

External interrupts are funneled through hardware interrupt controllers (such as the 8259A PIC or 80130 component). An individual master PIC can manage interrupts from as many as eight external sources. However, the iRMX 86 operating system also supports an expanded (or cascaded) environment in which up to seven input lines of one master PIC are connected to slave 8259A PICs. The eighth input line from the master PIC must be connected directly to the system clock. Since each of the slaves can manage eight interrupts, this allows the operating system to manage interrupts from as many as 56 external sources plus the system clock.

The interrupt lines of the master PIC and the interrupt lines of the slave 8259A PICs are associated with interrupt levels as shown in Figure 8-1. The master interrupt levels, numbered M0 through M7, correspond to interrupt vectors 56 through 63, respectively. The slave interrupt levels, numbered x0 to x7 (where x ranges from 0 to 7) correspond to interrupt vectors 64 through 127, respectively.

There are three restrictions you must obey when assigning interrupt levels to external sources. They are:

- You must assign the system clock to a master interrupt level. The level number is a configuration option and is described in the iRMX 86 CONFIGURATION GUIDE.

- When you attach an interrupting device to a level on the master PIC, you cannot also attach a slave PIC to the same level. For example, suppose that you physically attach the device to level M3. This means that Entry 59 (decimal) of the interrupt vector table must contain the address of the interrupt handler for the device. It also means that Entries 88 through 95 (decimal) of the interrupt vector table (the slave level entries that correspond to master level M3) will not be used.

- You cannot connect a slave PIC to master level M0 if an interrupting device connects directly to any other master level. Thus, if you assign the system clock to an interrupt level other than M0, you can connect at most six slave PICs to your master PIC. If you assign the system clock to level M0, you can connect seven slave PICs.

If your system is to include a 8087 Numeric Processor Extension, connect it to master level zero. In these cases, the system clock should be assigned to level M2 (this is the factory setting on most iAPX 86-based boards).

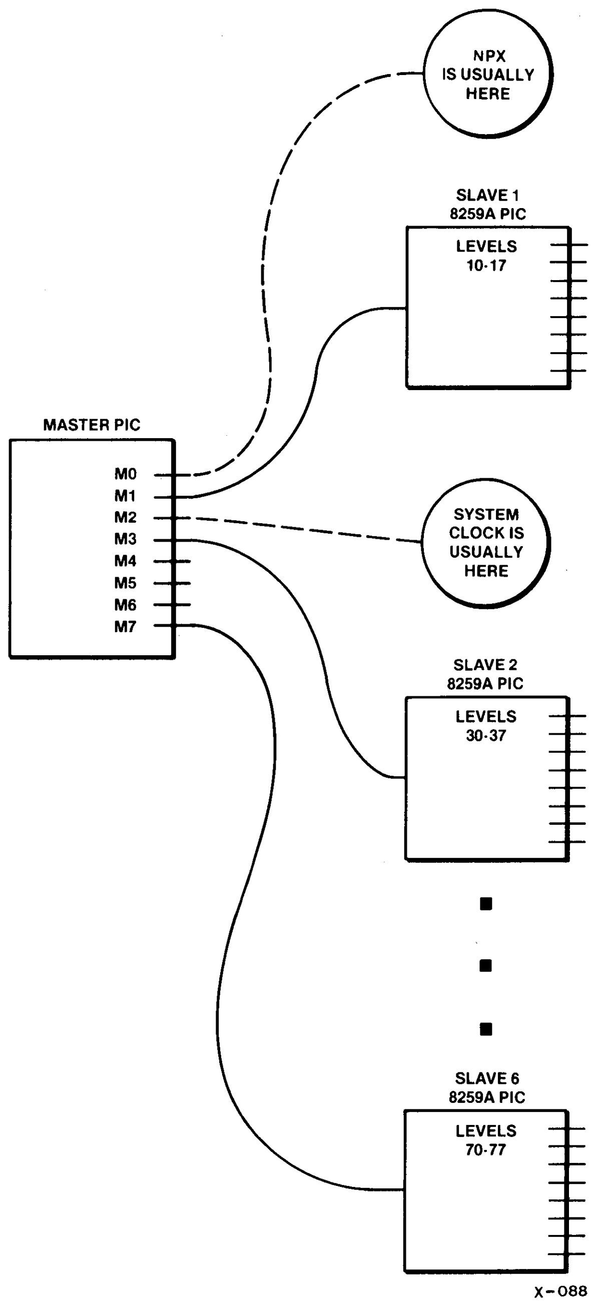


Figure 8-1. Cascaded Interrupt Levels

DISABLING INTERRUPTS

Occasionally you may want to prevent interrupt signals from causing an immediate interrupt. For example, it is desirable to prevent low priority interrupts from interfering with the servicing of a high priority interrupt. In the iRMX 86 Operating System, each interrupt level can be disabled. In some circumstances, described later, the Nucleus disables levels. Tasks can also disable and enable levels by means of the DISABLE and ENABLE system calls. The master level that you reserve for the system clock should not be disabled or enabled.

If an interrupt signal arrives at a level that is enabled, the Operating System transfers control to the address contained in the interrupt vector table entry that corresponds to the level on which the interrupt occurred. Otherwise, the level is disabled and the interrupt signal is blocked until the level is enabled, at which time the signal is recognized by the CPU. However, if the signal is no longer emanating from its source, it is not recognized and the interrupt is not handled.

There are four ways in which an interrupt level can be disabled.

- A task can explicitly disable a specific interrupt level by invoking the DISABLE system call. Later, a task can re-enable the level by invoking the ENABLE system call.

- Whenever a task invokes the SET$INTERRUPT system call, the task must specify a particular interrupt level. After the Operating System puts the start address of the interrupt handler into the appropriate entry of the interrupt vector table, the Operating System automatically enables the interrupt level.

  When a task invokes the SET$INTERRUPT system call to designate itself as the interrupt task for a particular interrupt level, the task can specify a limit to the number of interrupts that it will queue. If enough interrupts occur on the task's interrupt level, the queue can become full. Whenever this happens, the Operating System automatically disables the interrupt level until the queue ceases to be full.

- Whenever a task invokes the RESET$INTERRUPT system call to cancel the assignment of a particular interrupt handler to a particular interrupt level, the Operating System automatically disables the interrupt level.

- In order to provide preemptive priority-based scheduling, the Operating System can automatically disable or re-enable some interrupt levels whenever a task begins running, depending on the priority of the new running task and the priority of the previous running task. This allows high-priority tasks to run more quickly, without interrupts from lower-priority external devices. Table 8-1 shows the correlation between the levels disabled and the priority of the running task.

NOTE

A task should never use the PL/M-86 DISABLE statement or the ASM86 CLI (clear interrupt-enable flag) instruction to disable Operating System interrupts. The Nucleus does not guarantee that any interrupt level will still be disabled after the task invokes a Nucleus system call.

Table 8-1. Interrupt Levels Disabled for Running Task

Task Priority	Disabled Levels	
	Slave Levels	Master Levels
0-2	00 - 77	M0 - M7
3-4	01 - 77	M1 - M7
5-6	02 - 77	M1 - M7
7-8	03 - 77	M1 - M7
9-10	04 - 77	M1 - M7
11-12	05 - 77	M1 - M7
13-14	06 - 77	M1 - M7
15-16	07 - 77	M1 - M7
17-18	10 - 77	M1 - M7
19-20	11 - 77	M2 - M7
21-22	12 - 77	M2 - M7
23-24	13 - 77	M2 - M7
25-26	14 - 77	M2 - M7
27-28	15 - 77	M2 - M7
29-30	16 - 77	M2 - M7
31-32	17 - 77	M2 - M7
33-34	20 - 77	M2 - M7
35-36	21 - 77	M3 - M7
37-38	22 - 77	M3 - M7
39-40	23 - 77	M3 - M7
41-42	24 - 77	M3 - M7
43-44	25 - 77	M3 - M7
45-46	26 - 77	M3 - M7
47-48	27 - 77	M3 - M7
49-50	30 - 77	M3 - M7
51-52	31 - 77	M4 - M7
53-54	32 - 77	M4 - M7
55-56	33 - 77	M4 - M7
57-58	34 - 77	M4 - M7
59-60	35 - 77	M4 - M7
61-62	36 - 77	M4 - M7
63-64	37 - 77	M4 - M7
65-66	40 - 77	M4 - M7

Table 8-1. Interrupt Levels Disabled for Running Task (continued)

Task Priority	Disabled Levels	
	Slave Levels	Master Levels
67-68	41 - 77	M5 - M7
69-70	42 - 77	M5 - M7
71-72	43 - 77	M5 - M7
73-74	44 - 77	M5 - M7
75-76	45 - 77	M5 - M7
77-78	46 - 77	M5 - M7
79-80	47 - 77	M5 - M7
81-82	50 - 77	M5 - M7
83-84	51 - 77	M6 - M7
85-86	52 - 77	M6 - M7
87-88	53 - 77	M6 - M7
89-90	54 - 77	M6 - M7
91-92	55 - 77	M6 - M7
93-94	56 - 77	M6 - M7
95-96	57 - 77	M6 - M7
97-98	60 - 77	M6 - M7
99-100	61 - 77	M7
101-102	62 - 77	M7
103-104	63 - 77	M7
105-106	64 - 77	M7
107-108	65 - 77	M7
109-110	66 - 77	M7
111-112	67 - 77	M7
113-114	70 - 77	M7
115-116	71 - 77	None
117-118	72 - 77	None
119-120	73 - 77	None
121-122	74 - 77	None
123-124	75 - 77	None
125-126	76 - 77	None
127-128	77	None
129-255	None	None

INTERRUPT HANDLERS AND INTERRUPT TASKS

Whether an interrupt handler services an interrupt level by itself or invokes an interrupt task to service the interrupt depends on two factors:

- the kinds of system calls needed
- the amount of time required

Regarding the first factor, interrupt handlers can make only the ENTER$INTERRUPT, EXIT$INTERRUPT, GET$LEVEL, DISABLE and SIGNAL$INTERRUPT system calls. If the handler needs other system calls in order to service the interrupt, it must invoke an interrupt task.

Regarding the second factor, an interrupt handler should always invoke an interrupt task unless the handler can service interrupts quickly. This is because an interrupt signal disables all interrupts, and they remain disabled until the interrupt handler either services the interrupt or invokes an interrupt task. Invoking an interrupt task allows higher priority interrupts (and in some cases, the same priority interrupts) to be accepted.

SETTING UP AN INTERRUPT HANDLER

Interrupt handlers are generally written as PL/M-86 interrupt procedures, but can be written in assembly language. If an interrupt handler is written in assembly language, it must save and restore all register values, as noted later.

The SET$INTERRUPT system call binds an interrupt handler and, optionally, an interrupt task to an interrupt level. It does this as follows:

- One of the SET$INTERRUPT parameters, the interrupt$handler parameter, when used in conjunction with the PL/M-86 built-in function INTERRUPT$PTR, specifies the starting address of the interrupt handler. SET$INTERRUPT binds the handler to a level by placing this starting address into the interrupt vector table at the entry that corresponds to the level. When an interrupt of that level occurs, control automatically transfers through the vector table to the handler.

- Another parameter in SET$INTERRUPT, the interrupt$task$flag parameter, determines whether an interrupt task is associated with the level. If the interrupt$task$flag parameter is set to zero, there is no interrupt task for the specified level. Otherwise, the calling task becomes the interrupt task for the level.

Any desired value can be specified as the data segment base address for an interrupt handler by means of the interrupt$handler$ds parameter in SET$INTERRUPT. The interrupt handler can later cause this value to be loaded into the DS register by calling ENTER$INTERRUPT. This feature allows an interrupt handler and an interrupt task to share data areas.

When an iRMX 86 application system starts up, all interrupt levels are disabled. When SET$INTERRUPT binds an interrupt handler but not an interrupt task to a level, the level is enabled. If, instead, there is an interrupt task, the level is not enabled until that task makes a WAIT$INTERRUPT system call (described later.) An interrupt task should not enable its own level before making its first call to WAIT$INTERRUPT.

A RESET$INTERRUPT system call cancels the bond between an interrupt level and its interrupt handler. The call also disables the specified level. If there is an interrupt task for the level, RESET$INTERRUPT deletes it. DELETE$TASK does not delete interrupt tasks.

USING AN INTERRUPT HANDLER

If an interrupt handler is to service interrupts for a given level without invoking an interrupt task, the handler must assume one of two forms, depending on whether it needs to have the Nucleus set up its data segment base address.

If the interrupt handler does not need to access the data segment or if it contains its data segment base address in its code, then it should perform the following functions in the following order:

1. If in assembly language, save all register contents

2. Service the interrupt

3. Call EXIT$INTERRUPT

4. If in assembly language, restore all register contents

5. Return

The call to EXIT$INTERRUPT sends an end-of-interrupt signal to the hardware.

In contrast, if the interrupt handler wants the Nucleus to load a data segment base address (as specified in an earlier call to SET$INTERRUPT) into the DS register, then it should perform the following functions in the following order:

1. If in assembly language, save all register contents

2. Optionally, do some interrupt servicing

3. Call ENTER$INTERRUPT

4. Complete interrupt servicing

5. Call EXIT$INTERRUPT

6. If in assembly language, restore all register contents

7. Return

The call to ENTER$INTERRUPT tells the Nucleus to load the interrupt handler's data segment base address into the DS register. Because PL/M-86 makes use of the data segment, as specified by the contents of the DS register, loading a new value into this register serves to protect the data segment of the interrupted task.

USING AN INTERRUPT TASK

If there is both an interrupt handler and an interrupt task associated with a level, the interrupt handler invokes the interrupt task by making a SIGNAL$INTERRUPT system call. If a level has only an interrupt handler, however, the handler may not call SIGNAL$INTERRUPT.

If an interrupt handler invokes an interrupt task, the handler must perform the following functions in the following order:

1. If in assembly language, save the register contents.
2. Optionally, do some servicing.
3. Optionally, call ENTERINTERRUPT.
4. Optionally, begin servicing the interrupt without system calls.
5. Optionally, call EXIT$INTERRUPT or call SIGNAL$INTERRUPT.
6. Optionally, do some servicing.
7. If in assembly language, restore the register contents.
8. Return

The call to SIGNAL$INTERRUPT starts up the interrupt task, enables higher (and possibly equal) priority interrupts, and sends an End-of-Interrupt signal to the interrupt controller.

If used, the call to ENTER$INTERRUPT sets up a new DS value for the interrupt handler. If you want the interrupt handler to have the same DS value as that used by the interrupt task, so the handler can pass data to the task, follow the advice given in the description of the interrupt$handler$ds parameter of SET$INTERRUPT in Chapter 12.

An interrupt handler uses the resources of the interrupted task. The interrupt task, however, like any other task, has its own resources.

An interrupt task must perform the following functions in the following order, although the first two functions may be interchanged:

```
1.  Call SET$INTERRUPT.
2.  Do initialization.
3.  Do forever;
         Call WAIT$INTERRUPT.
         Service the interrupt (system calls allowed).
4.  End;
```

An interrupt task, once initialized, is always in one of two modes. Either it is servicing an interrupt or it is waiting for notification of an interrupt.

When a task becomes an interrupt task by calling SET$INTERRUPT, the Nucleus assigns a priority to it, according to the level that the task is to service. Table 8-2 shows the relationship between levels and interrupt task priorities.

NOTE

The priority that the Nucleus assigns to an interrupt task might exceed the maximum priority attribute of the job that contains that task. If this occurs, you get an exceptional condition. You should make sure this problem doesn't occur by creating the job with an appropriately high maximum priority attribute.

Table 8-2. The Relationship Between External Levels and Internal Task Priorities

LEVEL	INTERRUPT TASK PRIORITY	LEVEL	INTERRUPT TASK PRIORITY	LEVEL	INTERRUPT TASK PRIORITY
M0	18	20	36	50	84
M1	34	21	38	51	86
M2	50	22	40	52	88
M3	66	23	42	53	90
M4	82	24	44	54	92
M5	98	25	46	55	94
M6	114	26	48	56	96
M7	130	27	50	57	98
00	4	30	52	60	100
01	6	31	54	61	102
02	8	32	56	62	104
03	10	33	58	63	106
04	12	34	60	64	108
05	14	35	62	65	110
06	16	36	64	66	112
07	18	37	66	67	114
10	20	40	68	70	116
11	22	41	70	71	118
12	24	42	72	72	120
13	26	43	74	73	122
14	28	44	76	74	124
15	30	45	78	75	126
16	32	46	80	76	128
17	34	47	82	77	130

Figure 8-2 illustrates the two interrupt servicing patterns and their relationships.

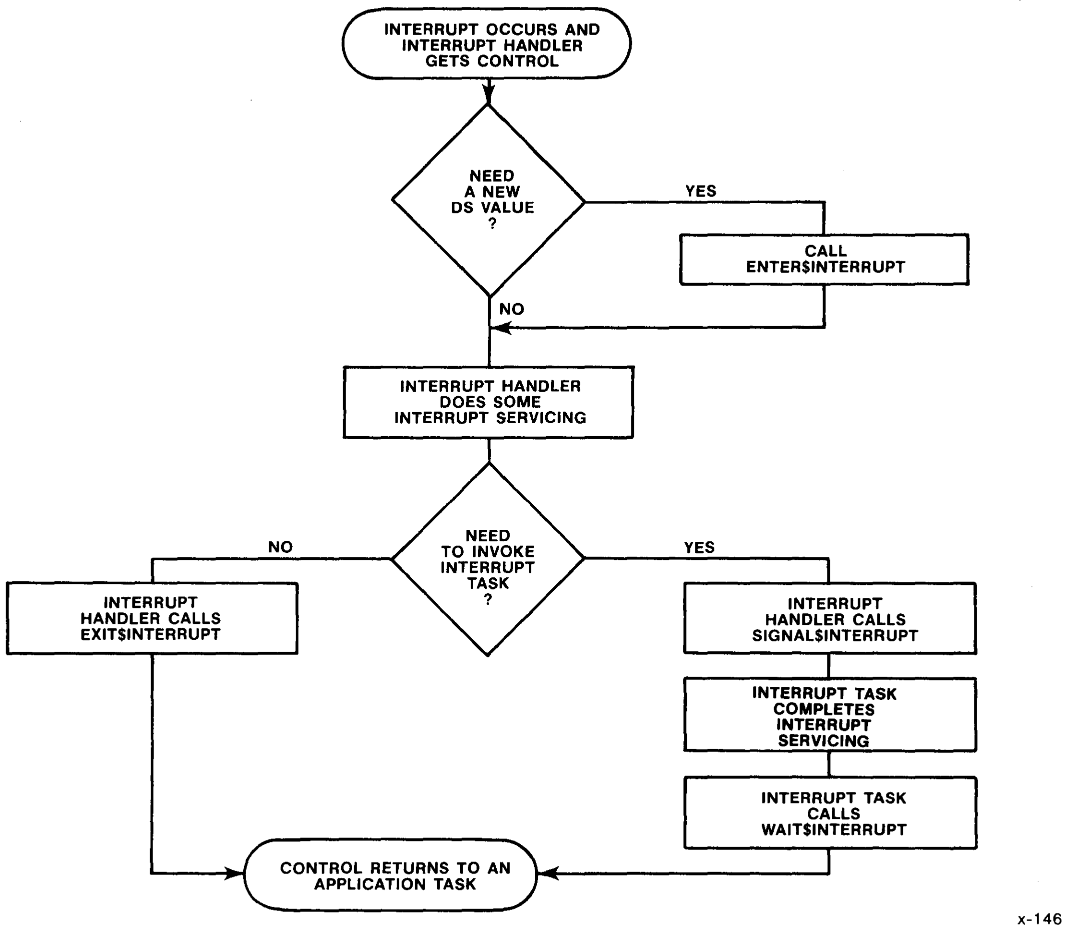

INTERRUPT OCCURS AND INTERRUPT HANDLER GETS CONTROL
NEED A NEW DS VALUE ?
YES
CALL ENTER$INTERRUPT
NO
INTERRUPT HANDLER DOES SOME INTERRUPT SERVICING
NO
NEED TO INVOKE INTERRUPT TASK ?
YES
INTERRUPT HANDLER CALLS EXIT$INTERRUPT
INTERRUPT HANDLER CALLS SIGNAL$INTERRUPT
INTERRUPT TASK COMPLETES INTERRUPT SERVICING
INTERRUPT TASK CALLS WAIT$INTERRUPT
CONTROL RETURNS TO AN APPLICATION TASK
x-146

Figure 8-2. Flow Chart of Interrupt Handling

NOTE

Because the automatic filling of the interrupt vector is overridden by the Nucleus, the NOINTVECTOR control should be used when compiling the interrupt handler.

Note that an interrupt handler might call an interrupt task sometimes yet not call it at other times. An example is an interrupt handler that puts characters entered at a terminal into a buffer. Whenever a character is received, the interrupt handler is invoked and puts the character in the line buffer. If the characater is an end-of-line character, or if the character count maintained by the interrupt handler indicates that the buffer is full, the interrupt handler calls its interrupt task to process the contents of the buffer. Otherwise, the interrupt handler calls EXIT$INTERRUPT and then returns control to application tasks. The next section discusses this kind of interrupt servicing in more detail.

USING MULTIPLE BUFFERS TO SERVICE INTERRUPTS

In certain instances, as illustrated in Figure 8-2, both an interrupt handler and an interrupt task are involved in servicing interrupts. The handler performs the simple, less time-consuming functions and then signals an interrupt task to perform more complicated functions. In doing this, the handler and the task usually exchange information by sharing data buffers. The handler places information into the buffers and the task uses that information. The number of buffers used determines when and how interrupts should be disabled.

Single Buffer Example

An example of a single buffer interrupt service mechanism is an interrupt handler that reads data from an external device character by character and places the characters into a buffer. When the buffer gets full, the handler calls SIGNAL$INTERRUPT to signal an interrupt task to further process the data. Since there is only one buffer for the data, the interrupt level associated with the interrupt task must be disabled while the task is processing. The Operating System, knowing (as a result of your task calling SET$INTERRUPT) that there is only one buffer, automatically disables the interrupt level when the handler invokes the SIGNAL$INTERRUPT system call. This prevents the interrupt handler from destroying the contents of the buffer by continuing to place data into an already full buffer. Figure 8-3 illustrates this situation.

Many users require only single buffering in their interrupt servicing routines. These users do not have to read the remaining paragraphs in this section. They should just ensure that their interrupt tasks specify a value of 1 for the interrupt$task$flag parameter in the call to SET$INTERRUPT. However, users who require multiple buffering for their interrupt servicing routines should continue reading this section.

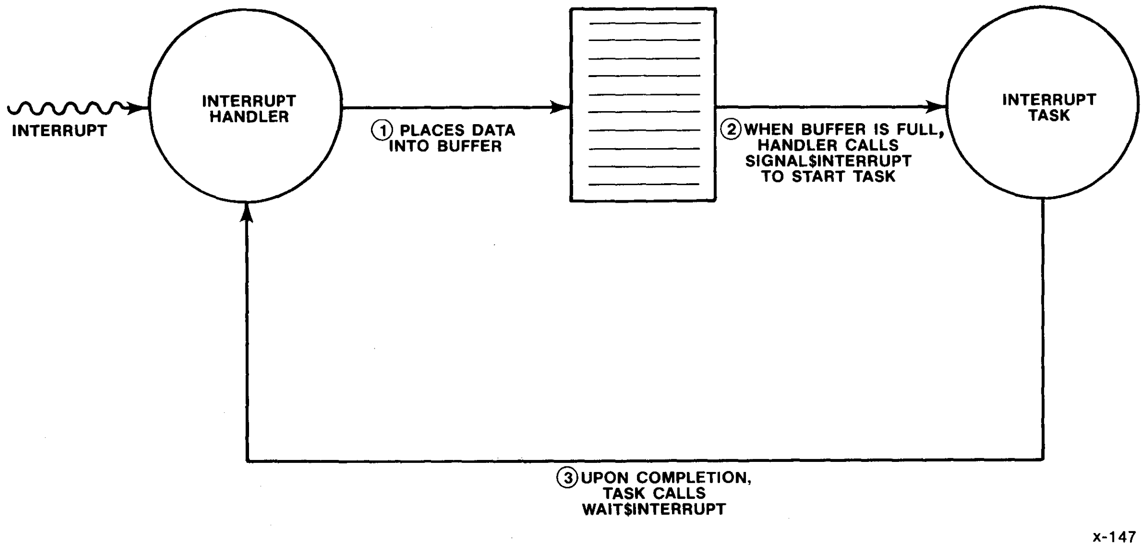

INTERRUPT
INTERRUPT HANDLER
① PLACES DATA INTO BUFFER
②WHEN BUFFER IS FULL, HANDLER CALLS SIGNAL$INTERRUPT TO START TASK
INTERRUPT TASK
③UPON COMPLETION, TASK CALLS WAIT$INTERRUPT
x-147

Figure 8-3. Single-Buffer Interrupt Servicing

Multiple Buffer Example

Now suppose that the interrupt handler and the interrupt task provide the same functions as in the first example, but use multiple buffers. In this case, the interrupt level associated with the task does not always have to be disabled while the task runs. Instead, the task can process a full buffer while the handler continues to accept interrupts. When the handler fills a buffer, it calls SIGNAL$INTERRUPT to start the interrupt task, as in the first example. However, because there are multiple buffers, the interrupt level is not disabled. Instead, the handler continues to accept interrupts, placing the data into the next empty buffer.

While this is going on, the interrupt task processes the full buffer. When the task completes the processing, it calls WAIT$INTERRUPT, to indicate that it is ready to accept another SIGNAL$INTERRUPT request (another full buffer) and to indicate that the buffer it just finished processing is available for reuse by the handler. Figure 8-4 illustrates this multiple buffer situation.

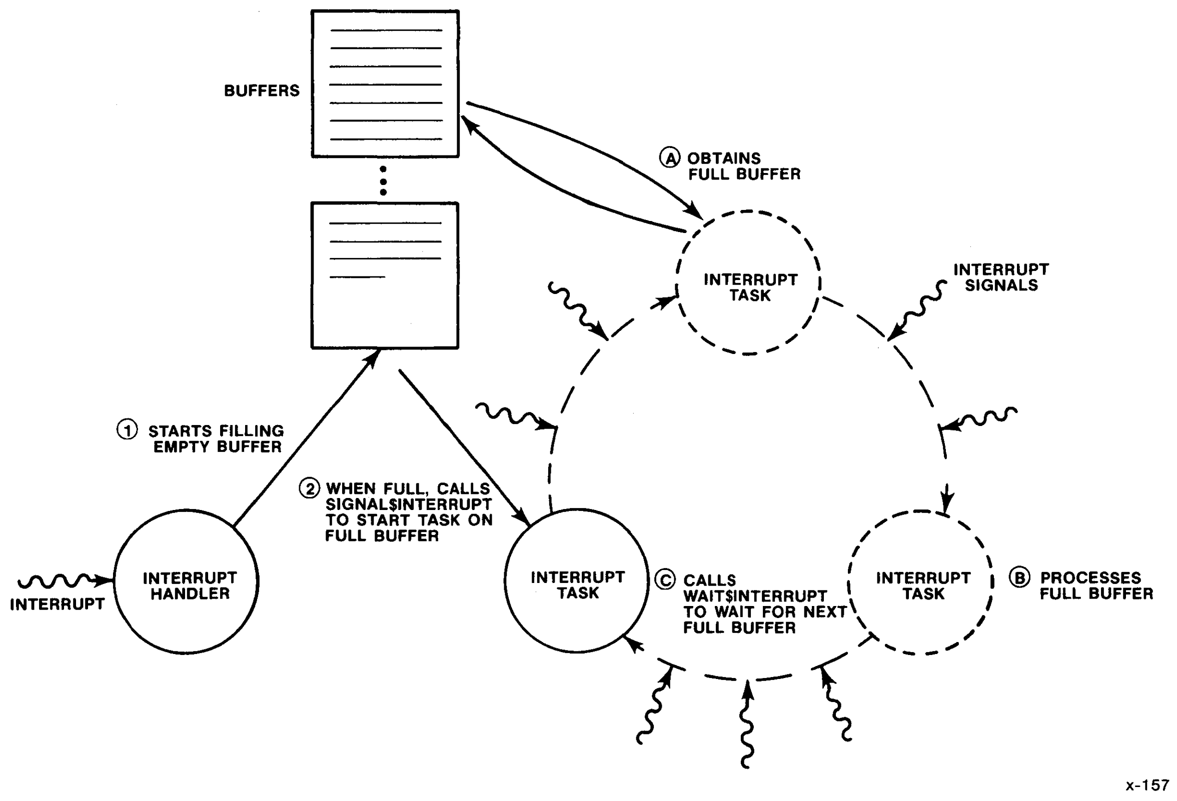

BUFFERS
Ⓐ OBTAINS FULL BUFFER
INTERRUPT TASK
INTERRUPT SIGNALS
① STARTS FILLING EMPTY BUFFER
② WHEN FULL, CALLS SIGNAL$INTERRUPT TO START TASK ON FULL BUFFER
INTERRUPT
INTERRUPT HANDLER
INTERRUPT TASK
Ⓒ CALLS WAIT$INTERRUPT TO WAIT FOR NEXT FULL BUFFER
INTERRUPT TASK
Ⓑ PROCESSES FULL BUFFER
x-157

Figure 8-4. Multiple-Buffer Interrupt Servicing

Because the handler and the task are running somewhat independently, the handler may fill a buffer and call SIGNAL$INTERRUPT before the task has finished processing the previous buffer. To prevent the SIGNAL$INTERRUPT request from becoming lost, the operating system maintains a count of these requests. Each time the handler calls SIGNAL$INTERRUPT, the count is incremented by one. Each time the task calls WAIT$INTERRUPT, the count is decremented by one.

If the count is still greater than zero after the interrupt task calls WAIT$INTERRUPT, the task does not wait for the next SIGNAL$INTERRUPT to occur before resuming execution. Instead, it realizes that outstanding requests exist and immediately starts processing the next request (the next full buffer). Thus, with proper tuning, neither the interrupt task nor the interrupt handler has to wait for the other. The interrupt handler can continually respond to interrupts without having the task disable the interrupt level. The interrupt task can continually process full buffers of data without waiting for the handler to call SIGNAL$INTERRUPT.

Specifying the Count Limit

The interrupt task, when it initially calls SET$INTERRUPT, puts a limit on the maximum number of outstanding SIGNAL$INTERRUPT requests. The interrupt$task$flag parameter specifies this limit. When the interrupt handler calls SIGNAL$INTERRUPT, causing the count to be incremented to the limit, two things happen. They are:

- The interrupt level is disabled, preventing the handler from accepting further interrupts until the interrupt task makes its next WAIT$INTERRUPT call.
- The E$INTERRUPT$SATURATION condition code is returned by SIGNAL$INTERRUPT to the handler, to indicate that the limit has been reached. This is an informative message only.

When the task calls WAIT$INTERRUPT and decrements the count below the limit, the interrupt level is enabled, allowing the handler to resume accepting interrupts.

The task should always set the limit equal to the number of buffers that the task and handler use. If the task sets the limit larger than the number of buffers, the handler will accept interrupts when no buffers are available and data will be lost. If the task sets the limit smaller than the number of buffers, there will always be empty buffers and space will be wasted.

For example, if one buffer is used, the task should set the limit to one. In this case, the interrupt level is always disabled while the task is processing the buffer. If two buffers are used, the task should set the limit to two. Then, the handler can fill one buffer while the task is processing the other. Additional buffers require correspondingly higher limits. However, if the task sets the limit to zero, the interrupt handler operates without an interrupt task.

NOTE

When an interrupt task sets the count limit to one, SIGNAL$INTERRUPT will not return the E$INTERRUPT$SATURATION condition code.

Table 8-3 illustrates the situation described in this section. It shows the actions of the handler and the task illustrated in Figure 8-3. The table is broken up into three parts: actions of the interrupt handler, actions of the interrupt task, and SIGNAL$INTERRUPT count. The count limit is set to two. The table shows the actions of both the handler and the task through time, and the change in value of the count.

Table 8-3 documents two extreme conditions, labeled "A" and "B". At position "A", the interrupt handler fills its last available buffer and calls SIGNAL$INTERRUPT to notify the task. However, at this point the task is not finished processing the first buffer. The count is incremented to the limit and interrupts are disabled until the task finishes with the first buffer and calls WAIT$INTERRUPT.

At position "B", the opposite case exists. The task finishes processing its buffer and calls WAIT$INTERRUPT. However, the handler has not processed enough interrupts to fill a buffer. The task must wait until the handler calls SIGNAL$INTERRUPT.

Table 8-3. Handler and Task Interraction Through Time

	Interrupt Handler	Interrupt Task	SIGNAL$ INTERRUPT Count
Time		Call SET$INTERRUPT to establish handler and task for level, setting count limit to 2.	0
		Call WAIT$INTERRUPT to wait for first request from handler.	0
Intrpt ⇝	Process interrupt, start filling first buffer.		
Intrpt ⇝	Process interrupt, continue filling first buffer.		
• • •			

Table 8-3. Handler and Task Interraction Through Time (continued)

	Interrupt Handler	Interrupt Task	SIGNAL$ INTERRUPT Count
Intrpt ⤳	Process interrupt. Buffer is full. Call SIGNAL$INTERRUPT. →	Start processing first full buffer.	1
Intrpt ⤳	Process interrupt. Start filling next buffer.		
⋮			
Intrpt ⤳ Ⓐ	Process interrupt. Buffer is full. Call SIGNAL$INTERRUPT. → Count is at limit. Interrupt level is disabled.		2
		Call WAIT$INTERRUPT. Task starts processing next full buffer immediately and returns empty buffer. Interrupt level is enabled.	1
Intrpt ⤳	Process interrupt. Start filling next buffer.		
⋮ Ⓑ		Call WAIT$INTERRUPT. No full buffers are available. Task waits for next request.	0
Intrpt ⤳	Process interrupt. Buffer is full. Call SIGNAL$INTERRUPT. →	Start processing next full buffer.	1
	⋮	⋮	⋮

Enabling Interrupt Levels From Within a Task

In certain cases, an interrupt task may finish with a buffer of data before it finishes its actual processing. An example of this is a task that processes a buffer and then waits at a mailbox, possibly for a message from a user terminal, before calling WAIT$INTERRUPT. If there are other buffers of data available to the handler (i.e. the count of outstanding SIGNAL$INTERRUPT requests has not reached the limit), this does not present a problem. The handler can continue accepting interrrupts and filling empty buffers. However, if the interrupt task is processing the last available buffer (i.e. the count limit has been reached), the interrupt handler cannot accept further interrupts, because the interrupt level is disabled. This may be an undesirable situation if the interrupt task takes a long time before calling WAIT$INTERRUPT.

To prevent this situation, the interrupt task can invoke the ENABLE system call immediately after it finishes with the buffer, to enable its associated interrupt level. This means that while the task engages in its time-consuming activities the interrupt handler can accept further interrupts and place the data into the buffer just released by the task.

However, if the interrupt handler fills the buffer and calls SIGNAL$INTERRUPT before the task calls WAIT$INTERRUPT, the following things occur:

- The count of outstanding SIGNAL$INTERRUPT requests is incremented, causing it to exceed the user-specified limit.
- An exception code, E$INTERRUPT$OVERFLOW, is returned to the interrupt handler to indicate this overflow.
- The interrupt level is again disabled. It cannot be enabled again until the count falls to or below the limit.

If the interrupt task calls ENABLE when the interrupt level is enabled or when the count is equal to the limit, nothing happens and no exception code is returned. However, if the interrupt task tries to enable the interrupt level when the count is greater than the limit, the ENABLE system call returns the E$CONTEXT exception code.

If a task other than an interrupt task tries to enable the level, one of three things can happen:

- If the level is already enabled, the ENABLE system call returns the E$CONTEXT condition code.
- If the non-interrupt task tries to enable the level (presumably following a DISABLE) and the interrupt task is not running (that is, the interrupt task has called WAIT$INTERRUPT and is waiting for a service request), the level is enabled immediately.
- If the interrupt task is running, the enable does not take effect until the interrupt task next invokes WAIT$INTERRUPT.

HANDLING SPURIOUS INTERRUPTS

When a PIC receives a signal from an interrupting device, it informs the Operating System of the interrupt level. If the interrupting device sends interrupt signals of short duration (that is, the input line is active for very short periods), the interrupt signal might be gone when the PIC tries to determine the interrupt level. If this happens, the PIC cannot determine the interrupt level and thus treats the interrupt as a spurious interrupt.

Each time the PIC detects a spurious interrupt, it responds as if a level 7 interrupt had occurred. So, if a master PIC detects a spurious interrupt, it responds as if the interrupt occurred on level M7. If a slave PIC detects a spurious interrupt (for example, a slave connected to master level M3), it responds as if the corresponding level 7 interrupt occurred (in this case, level 37).

A spurious interrupt indicates a problem; the PIC detected an interrupt signal but was unable to determine the level. Every application system should provide some means of isolating spurious interrupts so as to prevent further damage (such as falsely responding to a level 7 interrupt). This involves judiciously selecting interrupt levels and adding code to all level 7 interrupt handlers (handlers that service master level M7 or slave levels x7, where x ranges from 0 through 7). Once the spurious interrupt has been isolated, the level 7 interrupt handler can do one of two things:

- It can attempt to correct the problem.
- It can ignore the spurious interrupt and resume system processing.

In either case, before the handler returns control it should call EXIT$INTERRUPT to clear the hardware.

The following sections describe several options for isolating spurious interrupts.

CALLING GET$LEVEL

One way that a level 7 interrupt handler can check for spurious interrupts is by invoking the GET$LEVEL system call as soon as the handler starts running. GET$LEVEL returns the level of the highest priority interrupt which a handler has started but not yet finished processing. If the level returned is not the level associated with the interrupt handler, the interrupt is spurious.

This method is simple to implement, but it is a viable solution only for those handlers that can afford to spend the time required to execute GET$LEVEL. Some handlers may have speed requirements that prohibit the use of GET$LEVEL.

JUDICIOUS SELECTION OF INTERRUPT LEVELS

Another way to isolate spurious interrupts is to avoid connecting devices to level 7 interrupts (master level M7 and slave levels x7, where x ranges from 0 to 7). If you have no devices connected to these levels, and thus no handlers servicing them, spurious interrupts will not affect your system operation. Instead, each time a spurious interrupt occurs the PIC reacts as if a level 7 interrupt had occurred, sending control to interrupt vector table entry associated with the level 7 interrupt. But, because no handler is associate with that level, the vector table entry contains a pointer to the default handler, which returns control to the highest priority ready task.

EXAMINING THE IN-SERVICE REGISTER

Another way that a level 7 interrupt handler can check for spurious interrupts is by immediately reading the ISR (In-Service Register) of the PIC corresponding to the level. If the BYTE value obtained from that register does not have a 1 in the high-order bit, the interrupt is spurious. In order to read the value, the handler must know the port address of the ISR. In PL/M-86, the following lines perform this check when placed at the beginning of the interrupt handler:

```
    IF ((INPUT (port address of ISR)) AND 80H) = 0

        THEN interrupt is spurious
```

This method of isolating spurious interrupts should be used only as a last resort. It requires that the handler knows the address of the ISR (which may vary from system to system).

EXAMPLES OF INTERRUPT SERVICING

To help you understand the major points already described, Tables 8-4, 8-5, and 8-6 are provided. Each table outlines the turning points in a scenario where an interrupt handler is assigned to a level, an interrupt arrives at that level and is serviced, and finally the assignment of an interrupt handler is cancelled. Table 8-4 shows a case where the interrupt handler deals with the interrupt. Table 8-5 treats the case where the interrupt handler calls an interrupt task, either immediately or after filling a single buffer of data. Table 8-6 treats the case where an interrupt handler and an interrupt task use multiple buffers to service interrupts. Tables 8-4 and 8-5 assign the handler to master level 4. Table 8-6 assigns the handler to slave level 35.

In the right-hand column of each of tables 8-4, 8-5 and 8-6, the phrase "interrupt levels necessarily disabled" alludes to the fact that the events of the example cause certain levels to be enabled or disabled. Other events, outside the scope of the example, might cause other levels to be disabled as well.

Table 8-4. Servicing Interrupts with an Interrupt Handler

STEP	EVENTS	EXPLANATION	INTERRUPT LEVELS NECESSARILY DISABLED
1	-	No interrupt handler assigned to level M4.	M4
2	RQ$SET$INTERRUPT (LEVEL$4,0,...);	A task assigns an interrupt handler to level M4.	NONE
3	Level 4 device interrupts	An interrupt arrives at level M4.	M0-M7, 00-77
4	. . .	The interrupt is serviced by the interrupt handler.	M0-M7, 00-77
5	RQ$EXIT$INTERRUPT (LEVEL$4,...);	Interrupt hardware reset by the interrupt handler.	M0-M7, 00-77
6	Interrupt handler returns	Interrupts are re-enabled.	NONE
7	RQ$RESET$INTERRUPT (LEVEL$4,...);	A task cancels the assignment of an interrupt handler to level M4.	M4

Table 8-5. Servicing Interrupts with an Interrupt Task

STEP	EVENTS	EXPLANATION	INTERRUPT LEVELS NECESSARILY DISABLED
1	—	No interrupt handler assigned to level M4.	M4
2	RQ$SET$INTERRUPT (LEVEL$4, 1, ...);	A task assigns an interrupt handler to level M4 and it assigns itself to be the interrupt task for that level. It specifies that one SIGNAL$INTERRUPT request can be outstanding.	M4
3	RQ$WAIT$INTERRUPT (LEVEL$4,...);	The interrupt task begins to wait for an interrupt.	NONE
4	Level 4 device interrupts	An interrupt arrives at level M4. The interrupt handler gets control and optionally, does some servicing. The handler may service several interrupts by performing steps 4 through 6 of Table 8-4.	M0-M7, 00-77
5	RQ$SIGNAL$INTERRUPT (LEVEL$4,...);	The interrupt handler invokes the interrupt task.	M4-M7, 50-77
6	. . .	The interrupt is serviced by the interrupt task.	M4-M7, 50-57
7	RQ$WAIT$INTERRUPT (LEVEL$4,...);	The interrupt task finishes and begins to wait for another level M4 interrupt. Control passes back to the interrupt handler and then back to an application task.	NONE
8	RQ$RESET$INTERRUPT (LEVEL$4,...):	A task cancels the assignment of an interrupt handler to Level 4.	M4

Table 8-6. Servicing Interrupts with an Interrupt Handler, an Interrupt Task, and Multiple Buffering

STEP	EVENTS	EXPLANATION	INTERRUPT LEVELS NECESSARILY DISABLED
1	—	No interrupt handler assigned to level 35.	35
2	RQ$SET$INTERRUPT (LEVEL$35, 2, ...);	A task assigns an interrupt handler to level 35 and assigns itself to be the interrupt task for that level. It specifies that two SIGNAL$INTERRUPT requests can be outstanding (double buffering).	35
3	RQ$WAIT$INTERRUPT (LEVEL$35,...);	The interrupt task begins to wait for an interrupt.	NONE
4	Level 35 device interrupts	An interrupt arrives at level 35. The interrupt handler gets control and does some servicing.	M0-M7, 00-77
5	. . .	The handler services all interrupts, as described in steps 4 through 6 of Table 8-4, until the first buffer is full.	
6	RQ$SIGNAL$INTERRUPT (LEVEL$35,...);	The interrupt handler invokes the interrupt task.	M4-M7, 36-77
7	. . .	The interrupt task processes the full buffer. Meanwhile, the interrupt handler services interrupts, as described in steps 4 through 6 of Table 8-4, until the next buffer is full.	M4-M7, 36-77

Table 8-6. Servicing Interrupts with an Interrupt Handler, an Interrupt Task, and Multiple Buffering (continued)

STEP	EVENTS	EXPLANATION	INTERRUPT LEVELS NECESSARILY DISABLED
8	RQ$WAIT$INTERRUPT (LEVEL$35,...);	The interrupt task finishes and begins to wait for another signal from the interrupt handler. Control passes back to the interrupt handler and then back to an application task.	NONE
9	RQ$RESET$INTERRUPT (LEVEL$4,....);	A task cancels the assignment of an interrupt handler to Level M4.	M4

SYSTEM CALLS FOR INTERRUPTS

The following system calls manipulate interrupts:

- SET$INTERRUPT --- assigns an interrupt handler and, if desired, an interrupt task to an interrupt level.
- RESET$INTERRUPT --- cancels the assignment made to a level by SET$INTERRUPT and, if applicable, deletes the interrupt task for that level.
- EXIT$INTERRUPT --- used by interrupt handlers to send an end-of-interrupt signal to hardware.
- SIGNAL$INTERRUPT ---used by interrupt handlers to invoke interrupt tasks.
- WAIT$INTERRUPT --- suspends the calling interrupt task until it is called into service by an interrupt handler.
- ENABLE --- enables an external interrupt level.
- DISABLE --- disables an external interrupt level.

- GET$LEVEL --- returns the interrupt level of highest priority for which an interrupt handler has started but has not yet finished processing.
- ENTER$INTERRUPT --- sets up a previously designated data segment base address for the calling interrupt handler.

CHAPTER 9. REGIONS

The iRMX 86 Nucleus provides three types of exchanges: mailboxes, semaphores, and regions. Regions, unlike mailboxes and semaphores which are discussed in Chapter 4, should be restricted to special uses. Misuse of regions, which allow tasks to share data, can have profound affects on your application system.

RISKS INVOLVED IN SHARING DATA

Occasionally, several tasks in a system must share data. If the tasks run concurrently and the data is subject to change, access to the data must be restricted to one task at a time. The following example illustrates the importance of controlling tasks' access to data.

Suppose Tasks A and B are both part of an air-traffic-control application system. Task A runs at fixed time intervals and checks for any potential collisions. Task B runs as a result of an interrupt caused whenever the sweep of the radar detects an aircraft. Task B is of higher priority than Task A and is responsible for updating the position of the detected aircraft. Potentially, task B could corrupt the data used by Task A.

For instance, suppose that Task A is in the process of extrapolating the position of a particular aircraft. It first fetches the craft's last-reported position and uses the craft's velocity to estimate the position at some time in the near future. Suppose that Task A fetches the X-coordinate of the position and is preempted by Task B before fetching the Y- and Z-coordinates. Task B now updates the craft's X-, Y-, and Z-coordinates to reflect the fresh information gathered from the radar. Task B surrenders the processor, and the system resumes running Task A. Task A finishes fetching the craft's last-reported position but ends up with corrupt information. Instead of using (old X, old Y, old Z) or (new X, new Y, new Z), Task A believes the last reported position to be (old X, new Y, new Z). In this application, this error could lead to disaster.

Corruption of data can occur in this manner whenever the following three conditions are met:

- The data is shared between two or more tasks.
- The tasks sharing the data run concurrently. (In other words, one of the tasks could possibly preempt another.)
- At least one of the tasks changes the data.

Whenever all three of these conditions exist, you must take special precautions to protect the validity of the shared data. You must ensure that only one task has access to the shared data at any instant, and you must ensure that the task having access cannot be preempted by other tasks desiring access. This protocol for sharing data is called mutual exclusion.

MUTUAL EXCLUSION USING SEMAPHORES

As is discussed in the INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM, tasks can use semaphores to obtain mutual exclusion. However, using semaphores for this purpose can lead to two kinds of problems:

- Priority Bottlenecks

  Suppose that three tasks, Task A, B and C, have low, medium and high priority, respectively. If these tasks employ a priority-queued semaphore to ensure that no more than one of them uses shared data at any instant, the following situation could arise:

  1. Task A (low priority) obtains access to the data and continues to run.
  2. Task C (high priority) attempts to gain access, but is forced to wait at the semaphore until Task A frees the data.
  3. Task B (medium priority) awakens from a timed sleep and preempts Task A (low priority).

  In Step 2, Task C must wait for Task A (which has lower priority) to finish using the shared data. This is reasonable as Task A gained access to the data before Task C. This kind of delay is inherent in mutual exclusion.

  In Step 3, however, the delay is unreasonable. Task C is forced to wait for Task B (which has lower priority than Task C) even if Task B does not use the shared data.

- Tying Up the Shared Data

  If several tasks use a semaphore to govern access to shared data, and the task currently having access is suspended, the semaphore prevents any other tasks from using the shared data. Only after the suspended task is resumed can it free the shared data for use by the other tasks.

  If the task using the data is deleted, rather than merely being suspended, the situation is even worse. The governing semaphore prevents any other tasks from ever using the shared data.

You can eliminate both of these kinds of problems by using regions rather than semaphores to govern the sharing of data.

MUTUAL EXCLUSION USING REGIONS

A region is an iRMX 86 object that tasks can use to guard a specific collection of shared data. Each task desiring access to shared data awaits its turn at the region associated with that data. When the task currently using the shared data no longer needs access, it notifies the Operating System, which then allows the next task to access the shared data.

Noteworthy are the following facts regarding regions:

- The priority of the task that currently has access to the shared data may temporarily be raised. This happens automatically (at regions where the task queue is priority-based) whenever the task at the head of the queue has a priority higher than that of the task that has access. Under such circumstances, the priority of the task having access is raised to match that of the task at the head of the queue. When the task having access surrenders access, its priority automatically reverts to its original value. This priority adjustment prevents the priority bottleneck that can occur when tasks use semaphores to obtain mutual exclusion.

- Once a task gains access to shared data through a region, the task can not be suspended or deleted by other tasks until it surrenders access. This characteristic prevents tasks from tying up shared data.

CAUTION

When a task gains access through a region, it must not attempt to suspend or delete itself. Any attempt to do so will lock up the region, preventing other tasks from accessing the data guarded by the region. In addition, the task will never run again and its memory will not be returned to the memory pool. Also, if the task in the region attempts to delete itself, all other tasks that later attempt to delete themselves will encounter the same memory pool problems.

It should also be noted that you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application can not be stopped asynchronously (via CTRL/c entered at a terminal) while the task in the region.

- When you create a region you must specify which of two rules is to be used to determine which waiting task next gains access to the shared data. One rule is first-in/first-out (FIFO), and the other is priority.

- Regions are much faster than semaphores. The system calls used to manipulate a region require much less processor time than do those that manipulate semaphores.

USEFULNESS OF SEMAPHORES

After reading the last section, you are probably wondering why anyone would want to use semaphores at all. There are three reasons:

1. You can use semaphores to accomplish much more than mutual exclusion. For example, with semaphores you can synchronize multiple tasks or allocate resources. Regions, on the other hand, provide only mutual exclusion.

2. Because of the possibility of deadlock, regions should not be used outside of extensions to the Operating System. Consequently, programmers not familiar with Operating System extensions must use semaphores to accomplish mutual exclusion.

3. Semaphores allow a task to set an upper limit on the amount of time the task is willing to wait for access. In contrast, regions provide no such option. Tasks using regions for mutual exclusion have only two choices:

   - They can request immediate access. If a task makes such a request and access is not available immediately, the task does not wait at the region. Rather, it receives an exception code and continues to run.

   - They can request access as it becomes available. This kind of request causes the task to wait at the region until access becomes available. If access never becomes available, the task never runs again.

   Tasks use the ACCEPT$CONTROL system call to request immediate access. They use the RECEIVE$CONTROL system call to request access as it becomes available. Both of these system calls are described in detail in Chapter 12 of this manual.

REGIONS AND DEADLOCK

A major concern in any multitasking system is avoiding deadlock. Deadlock occurs when one or more tasks permanently lock each other out of required resources. The following hypothetical situation illustrates a method for quickly causing deadlock by using nested regions. An explanation of how to avoid the illustrated deadlock situation follows the example.

NOTE

> In the following example, the only system call used to gain access is the RECEIVE$CONTROL system call. Tasks using the ACCEPT$CONTROL system call cannot possibly deadlock at a region unless they keep trying endlessly to accept control.

Suppose that two tasks, A (high priority) and B (low priority), both need access to two collections of shared data. Call the two collections of data Set 1 and Set 2. Access to each set is governed by a region (Region 1 and Region 2).

Now suppose that the following events take place in the order listed:

1. Task B requests access to Set 1 via Region 1. Access is granted.

2. Before Task B can request access to Set 2, an interrupt occurs and Task A preempts Task B.

3. Task A requests access to Set 2 via Region 2. Access is granted.

4. Task A requests access to Set 1 via Region 1. Task A must wait because Task B already has access.

5. Task B resumes running and requests access to Set 2 via Region 2. Task B must wait because Task A already has access.

At this point Task A is waiting for Task B and vice versa. Tasks A and B are hopelessly deadlocked, and any other tasks that request access to either set of data will also become deadlocked.

This team deadlock situation applies only to systems in which regions are nested. If your system must use nested regions, you can prevent team deadlock by adhereing to the following rule:

Apply a strict ordering to all the regions in your system, and code tasks so that they gain access according to the order. For example, suppose that your system uses 12 regions. Write the names of the regions on a piece of paper in any order, and number them starting with 1. As you program a task that nests any of the regions (say Regions 3, 8, and 10), be sure that the task requests access in numerical order. The essential element of this technique is that all tasks must request access in a consistent order. The precise order is unimportant as long as all tasks obey it.

If you follow this rule consistently, you can safely nest regions to any depth.

REGIONS AND SYSTEM KNOWLEDGE

Use (and perhaps knowledge) of regions should be restricted to programmers that have a firm understanding of the Operating System and the entire application system. A careless or unscrupulous programmer can, by abusing regions, corrupt the interaction between tasks in an application system. For instance, by creating a region and gaining access to nonexistent shared data, unscrupulous programmers can make their tasks immune to deletion. If they never surrender access, the tasks can permanently avoid deletion.

Abusing some of the features described in this manual can affect the integrity of the entire Operating System. Regions constitute such a feature. If you wish to preserve the integrity of your application system, you should confine the use of regions to programmers who work with the Operating System and, even then, only within Operating System extensions.

SYSTEM CALLS FOR REGIONS

The following system calls manipulate regions:

- ACCEPT$CONTROL

  This system call allows a task to gain access to shared data only when access is immediately available. If a different task already has access, the requesting task remains ready but receives an exception code.

- CREATE$REGION

  This system call creates a region and returns a token for it. One of the parameters passed during this call specifies the queuing rule (FIFO or priority).

- DELETE$REGION

  This system call deletes a region.

- RECEIVE$CONTROL

  This system call causes a task to wait at the region until the task gains access to the shared data.

- SEND$CONTROL

  This system call, when issued by a task, frees the Operating System to grant a different task with access to the shared data.

CHAPTER 10. OPERATING SYSTEM EXTENSIONS

A feature of the iRMX 86 Operating System is that it can be extended to include your own customized objects and system calls. This feature allows you to create an operating system that precisely meets the needs of your application. This chapter explains how to extend the iRMX 86 Operating System to include your own system calls.

Material presented in this chapter is intended for programmers who write system programs to modify the Operating System.

THREE WAYS OF ADDING FUNCTIONALITY

Whenever more than one job in your application system requires a function not supplied by the iRMX 86 Operating System, you have at least the following three ways of adding the needed function:

- Write the function as a procedure and place it in a library by using LIB86. After compiling each job that requires the function, use LINK86 to link the library to the object module for the job.
- Write the function as a task and allow application tasks to invoke the function through a mailbox-segment interface.
- Write the function as a procedure and add it to the iRMX 86 Operating System. Application programs then invoke the function by means of a system call.

The relative advantages and disadvantages of the three alternatives are summarized in Table 10-1.

The third alternative involves extending the Operating System. The procedures that you must add to the Operating System in order to support the added function are called an Operating System extension, or OS extension. From the application programmer's standpoint, an OS extension appears to be a collection of one or more customized system calls.

CREATING AN OPERATING SYSTEM EXTENSION

Creating an OS extension involves both writing several procedures and initializing the interrupt vector of the iAPX 86 microprocessor.

Table 10-1. Comparison of Techniques for Creating Common Functions

	PROCEDURE LIBRARY	TASK	OS EXTENSION
INTERFACE FOR APPLICATION PROGRAMS	SIMPLE	COMPLEX	SIMPLE
RELATIVE PERFORMANCE	GOOD (for all functions)	POOR (for quick functions) MODERATE (for slower functions)	MODERATE (for quick functions) GOOD (for slower functions)
SYNCHRONOUS or ASYNCHRONOUS CALLS	BOTH	ASYNCHRONOUS ONLY	BOTH
SYSTEM PROGRAMMING	NOT REQUIRED	NOT REQUIRED	REQUIRED
DUPLICATE CODE	Difficult to avoid	Easy to avoid	Automatically avoided
REQUIRES RELINKING TO CHANGE	YES	NO	NO
SUPPORTS NEW OBJECT TYPES	NO	NO	YES

PROCEDURES USED IN OPERATING SYSTEM EXTENSIONS

Every OS extension is composed of at least two kinds of procedures. Figure 10-1 illustrates the simplest arrangement. The two required kinds of procedure are the following:

- Interface Procedure

  An interface procedure connects the customized system call to the Operating System. For example, to issue a NEW$FUNCTION system call, an application task executes a statement like

  ```
      CALL NEW$FUNCTION(......);
  ```

  This statement is, in fact, a call to an interface procedure, named NEW$FUNCTION, that transfers control to the Operating System. One interface procedure is required for each customized system call.

- Function Procedure

  The function procedure does the important work of the system call. That is, it performs the actions requested by the calling task. One function procedure is required for each customized system call.

Figure 10-1 depicts four OS extensions, each containing one system call. Note that the interface procedures are part of the application software and the function procedures are part of the system software. The tasks are linked to the interface procedures, but the interface procedures are not linked to the function procedures. Instead, the interface procedures pass control to the function procedures by way of the interrupt vector.

The interrupt vector consists of 256 four-byte entries; the first entry is at location 0 and the last is at location 1020 (decimal). The iRMX 86 Operating System uses these entries for many purposes, but the last 32 (entries 224 through 255) are reserved for user-supplied OS extensions.

In Figure 10-1, the four interface procedures transfer control to the four function procedures through four separate interrupt vector entries (each of which must be numbered in the 224 to 255 range). Note that, if confined to the pattern illustrated in Figure 10-1, a system is limited to 32 customized system calls.

In order to conserve system calls, another kind of procedure must be employed:

- Entry Procedure

  The entry procedure serves as a multiplexor for OS extensions supporting more than one system call. Figure 10-2 depicts a single OS extension with four system calls. The primary purpose of the entry procedure is to route the call from the interface procedure to the proper function procedure. Note that four interface procedures are still required to support the four system calls.

  The principal advantage of having an entry procedure is that one interrupt vector entry can support multiple system calls. This means that the 32 entries in the interrupt vector, along with entry procedures, can support a virtually unlimited number of customized system calls.

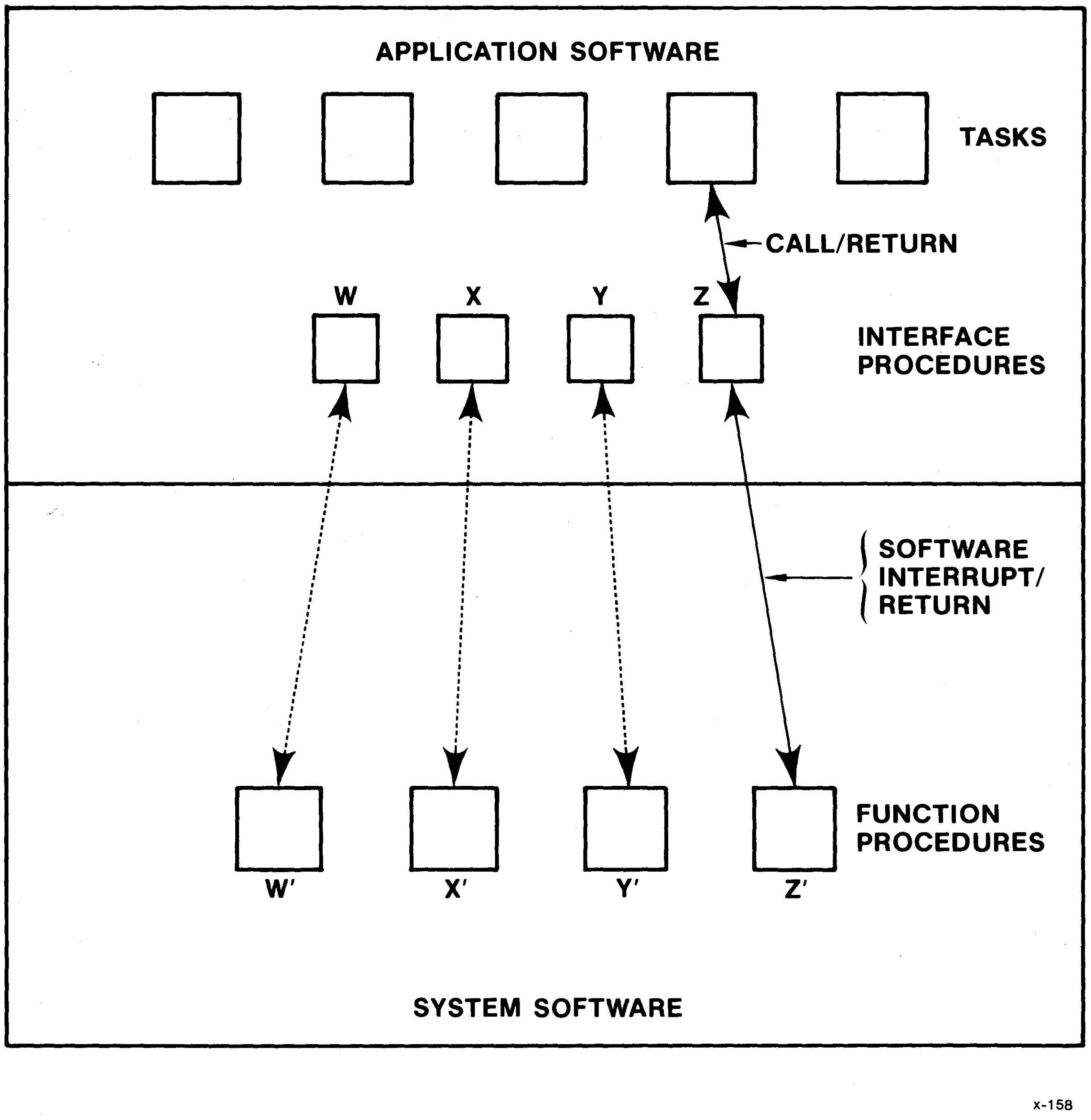


Figure 10-1. OS Extensions Without Entry Procedures

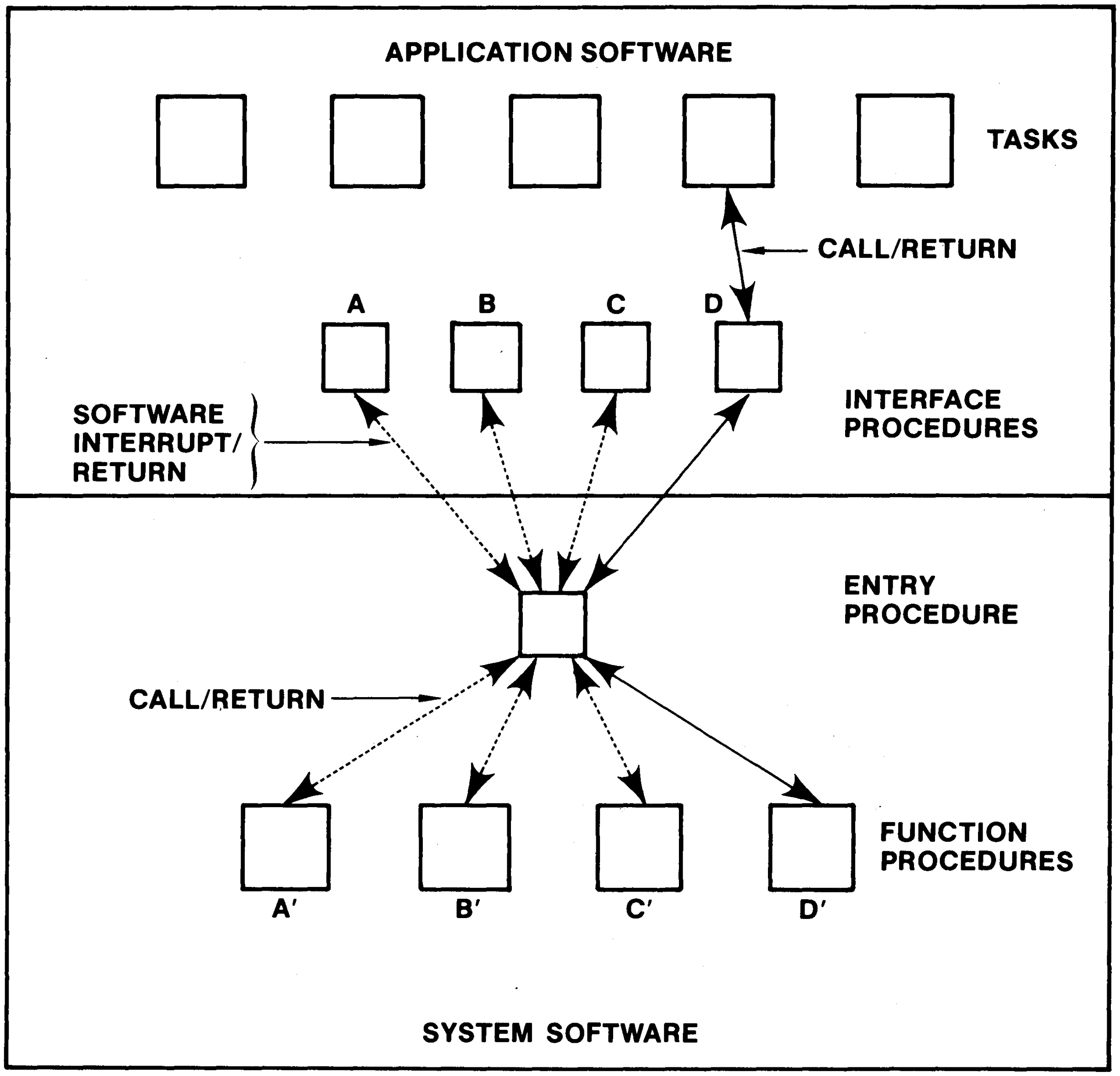


Figure 10-2. OS Extension with Procedure Entry

The following paragraphs describe the responsibilities of each of the kinds of procedures composing OS extensions. Figure 10-3 contains, in algorithmic form, summaries of these descriptions. Also, Chapter 11 contains an example of an OS extension that manages a customized object type.

Interface Procedures

For each system call in your OS extension, you must write a reentrant assembly language interface procedure. (For detailed information concerning the 8086 Asssmbly Language, refer to the appropriate 8086/8087/8088 MACRO ASSEMBLY LANGUAGE REFERENCE MANUAL.) The primary purpose of this procedure is to use a software interrupt to transfer control from the task that invoked the system call to an entry procedure (or, in the absence of an entry procedure, to a function procedure).

If there is an entry procedure, the interface procedure must communicate to it a code which identifies the function procedure that the entry procedure is to call. The interface procedure does this by loading the code into a previously-designated register or onto the stack of the calling task. The entry procedure, when invoked, extracts the code from this register or the stack.

A second important function of an interface procedure is informing the calling task (or its exception handler) of any exceptional conditions that have occurred. The entry procedure (or the function procedure if no entry procedure exists) communicates this information to the interface procedure by placing the exception code in the CX register and the number of the parameter that caused the error in the DL register. The interface procedure then does the following:

- Checks the CX register for the condition code. If this register contains a value other than zero (E$OK), an exceptional condition exists.
- If an exceptional condition exists, calls a procedure named RQ$ERROR.

The Nucleus interface library contains a default RQ$ERROR procedure. This procedure gets the exception code and parameter number from the CX and DL registers and then makes a SIGNAL$EXCEPTION system call to inform the calling task (or its exception handler) of the exception. When SIGNAL$EXCEPTION returns to the RQ$ERROR procedure, RQ$ERROR restores CX and DL with the exception code and parameter number and places a value of OFFFFH in the AX register.

If you do not want to use this default procedure, you can write your own RQ$ERROR procedure. Your RQ$ERROR procedure can perform any functions it needs in order to inform the application task of the exceptional condition. The only restriction placed on an RQ$ERROR procedure is that it should always return a value of OFFFFH in the AX register (so that OFFFFH is returned as a function value for your system calls that are typed procedures). An example of an alternate RQ$ERROR procedure is one that simply places OFFFFH in AX and then does a RETURN, returning control directly to the application task to avoid the task's normal exception handler.

To make sure that your own RQ$ERROR procedure is called instead of the default version, you should link your procedure directly to the interface procedure or include it in a library with the rest of your interface procedures. When linking your modules together, this library should always precede the Nucleus interface library in the link sequence.

Another important purpose of interface procedures is that they compensate, on behalf of the entry or function procedures that they call, for differences between parameter-passing protocols. Three different models (COMPACT, MEDIUM, and LARGE) are available when compiling iRMX 86 tasks written in PL/M-86. (Refer to the PL/M-86 COMPILER OPERATING INSTRUCTIONS FOR 8080/8085-BASED DEVELOPMENT SYSTEMS manual for information regarding these models.) By providing a library of interface procedures for each PL/M-86 model, you make the entry and function procedures independent of the PL/M-86 model in which application code is being compiled. If other languages were available, the same strategy would make the entry and function procedures independent of the language in which application code is written. The benefit of this independence is that only one entry procedure (or function procedure, if no entry procedure exists) is needed for each interrupt vector entry in your system.

Entry Procedures

Each OS extension comprising more than one system call must include a reentrant entry procedure, whose chief purpose is to route the call to the appropriate function procedure. Other duties of entry procedures are the following:

- Set up the exception handling mechanism for the OS extension. This can be done in one of two ways, depending on whether the OS extension has its own exception handler or whether it wants to handle exceptions in-line.

  If the OS extension has its own exception handler, the entry procedure must change the exception handler from that of the calling task to an exception handler for the OS extension. It must do this to guarantee that an error by the OS extension doesn't cause the calling task to be deleted (a common function of exception handlers). To make this change, the entry procedure calls GET$EXCEPTION$HANDLER to obtain and save the task's exception handler address and exception mode. It then calls SET$EXCEPTION$HANDLER to set new values for these entities. When control returns to the entry procedure from the function procedure, the entry procedure again calls SET$EXCEPTION$HANDLER to restore the original values.

  If you want the OS extension to handle its exceptions in-line, you must create your own RQ$ERROR procedure and link it to the entry procedure. This RQ$ERROR procedure must return control directly to the entry procedure instead of calling SIGNAL$EXCEPTION. If you supply an RQ$ERROR procedure of this type, the entry procedure does not have to change exception handlers. Instead, if the OS extension encounters exceptional conditions while invoking other system calls, this RQ$ERROR procedure is called to return control directly to the procedure that incurred the error. That procedure can then handle the error. It does not matter which exception handler is associated with the application task, since the exception handler is not called. The RQ$ERROR procedure is discussed in more detail later in this chapter.

- Perform any chore required by all system calls in this OS extension. By performing common chores in the entry procedure, you can factor code out of several function procedures.

- If notified by the function procedure that an exception occurred which must be transmitted back to the application task, do the following:

    Place the exception code in the CX register.

    Place the number of the parameter that caused the exceptional condition in the DL register.

    Return control to the interface procedure.

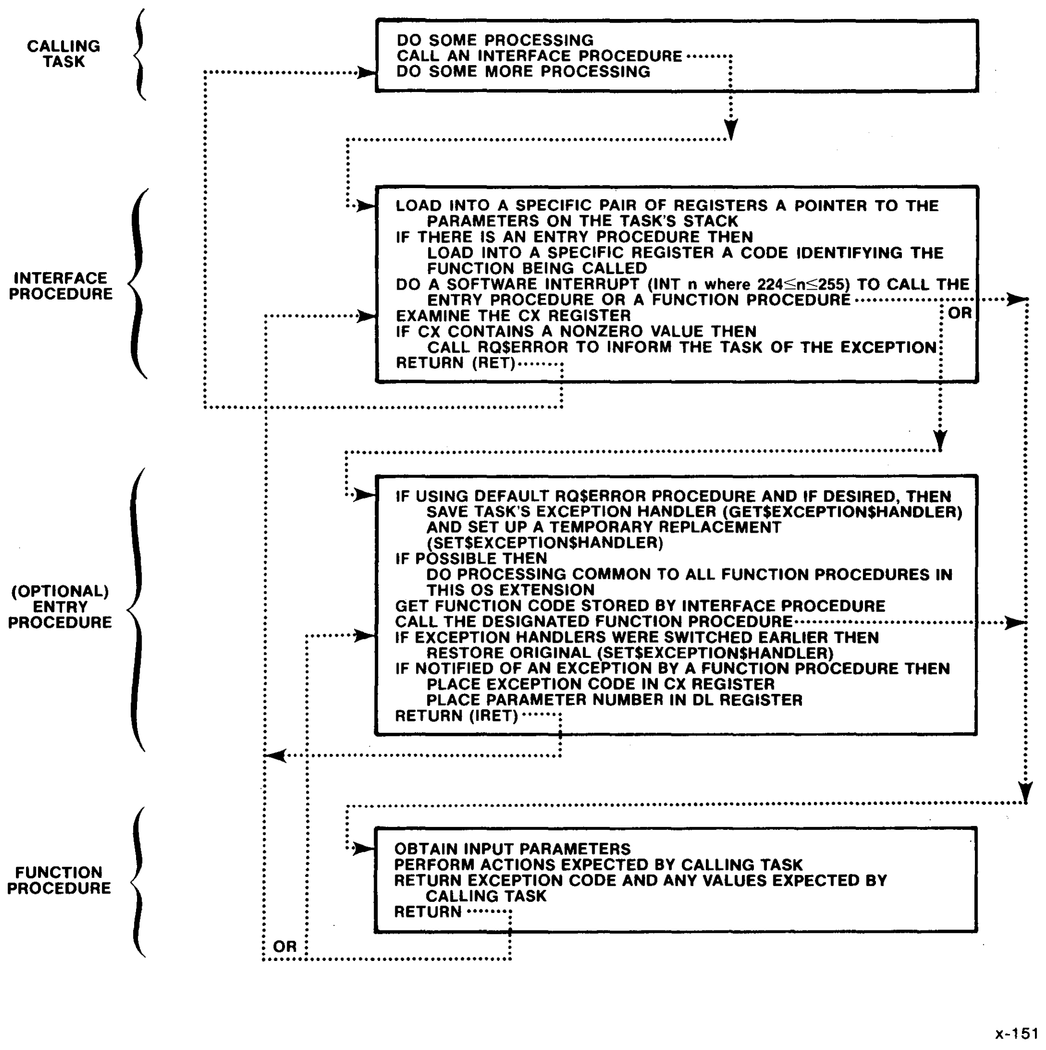
  The interface procedure should examine the CX register to check for an exceptional condition and call the version of RQ$ERROR to which it is linked.

  When adding OS extensions, you might wish to add your own customized exceptional conditions and associated codes. Values available to users for exception codes are 4000H to 7FFFH (for environmental conditions) and 0C000H to 0FFFFH (for programmer errors).

Write the entry procedure in assembly language so that you can directly access the stack and the registers. This provides you with the following benefits:

- It gives you access to the input parameters passed by the calling task and the interface procedure.

- It allows you to set the CX and DL registers in the event of an exceptional condition.



```
DO SOME PROCESSING
CALL AN INTERFACE PROCEDURE
DO SOME MORE PROCESSING
```

```
LOAD INTO A SPECIFIC PAIR OF REGISTERS A POINTER TO THE
    PARAMETERS ON THE TASK'S STACK
IF THERE IS AN ENTRY PROCEDURE THEN
    LOAD INTO A SPECIFIC REGISTER A CODE IDENTIFYING THE
    FUNCTION BEING CALLED
DO A SOFTWARE INTERRUPT (INT n where 224≤n≤255) TO CALL THE
    ENTRY PROCEDURE OR A FUNCTION PROCEDURE
EXAMINE THE CX REGISTER
IF CX CONTAINS A NONZERO VALUE THEN
    CALL RQ$ERROR TO INFORM THE TASK OF THE EXCEPTION
RETURN (RET)
```

```
IF USING DEFAULT RQ$ERROR PROCEDURE AND IF DESIRED, THEN
    SAVE TASK'S EXCEPTION HANDLER (GET$EXCEPTION$HANDLER)
    AND SET UP A TEMPORARY REPLACEMENT
    (SET$EXCEPTION$HANDLER)
IF POSSIBLE THEN
    DO PROCESSING COMMON TO ALL FUNCTION PROCEDURES IN
    THIS OS EXTENSION
GET FUNCTION CODE STORED BY INTERFACE PROCEDURE
CALL THE DESIGNATED FUNCTION PROCEDURE
IF EXCEPTION HANDLERS WERE SWITCHED EARLIER THEN
    RESTORE ORIGINAL (SET$EXCEPTION$HANDLER)
IF NOTIFIED OF AN EXCEPTION BY A FUNCTION PROCEDURE THEN
    PLACE EXCEPTION CODE IN CX REGISTER
    PLACE PARAMETER NUMBER IN DL REGISTER
RETURN (IRET)
```

```
OBTAIN INPUT PARAMETERS
PERFORM ACTIONS EXPECTED BY CALLING TASK
RETURN EXCEPTION CODE AND ANY VALUES EXPECTED BY
    CALLING TASK
RETURN
```

Figure 10-3. Summary of Duties of Procedures in OS Extensions

Function Procedures

The duties of the function procedure are principally to perform the actions requested by the calling task. Additionally, if there is not an entry procedure, the function procedure should inform the interface procedure concerning the exception status of the call. It should do this by setting CX and DL as described previously in the description of entry procedures. Function procedures should be reentrant and can be written in PL/M-86 or assembly language.

RQ$ERROR Procedures

The sections of this chapter that describe interface procedures and entry procedures both make mention of a procedure named RQ$ERROR. This is a procedure called by the interface procedures of the Nucleus and each subsystem of the Operating System in the event of an exceptional condition. For example, if your application task makes a SEND$MESSAGE system call and an exceptional condition results, the Nucleus returns the error (in the CX and DL registers as described previously) to the Nucleus interface library that is linked to your application task. The procedure in the library then calls RQ$ERROR to process the error.

This is not only true for application tasks that make system calls, but also for Intel-supplied subsystems (such as the I/O System) and OS extensions that make system calls. For example, if the I/O System calls SEND$MESSAGE and an exceptional condition results, the Nucleus returns the error to the Nucleus interface library that is linked to the I/O System. The procedure in that library calls RQ$ERROR to process the error.

Every subsystem of the Operating System that implements system calls also provides this mechanism for returning exceptions. If an application task makes an I/O system call (CREATE$FILE, for example) and incurs an exceptional condition, the I/O System returns control to the I/O System interface library that is linked to that task. The interface procedure in that library calls RQ$ERROR to process the error.

The OS extensions you write should also provide this mechanism for returning exceptions to tasks (or other OS exceptions) that invoke your customized system calls. The previous sections of this chapter describe the method for doing this.

The Nucleus interface library, as released, contains a default RQ$ERROR procedure. The function of this RQ$ERROR procedure is to call SIGNAL$EXCEPTION to inform the calling task (or its exception handler) of the exception. This version of RQ$ERROR should be linked to application tasks to ensure that their exception handlers are called when exceptional conditions occur. Figure 10-4 illustrates the flow of control from an application task to an exception handler when the task incurs an exceptional condition.

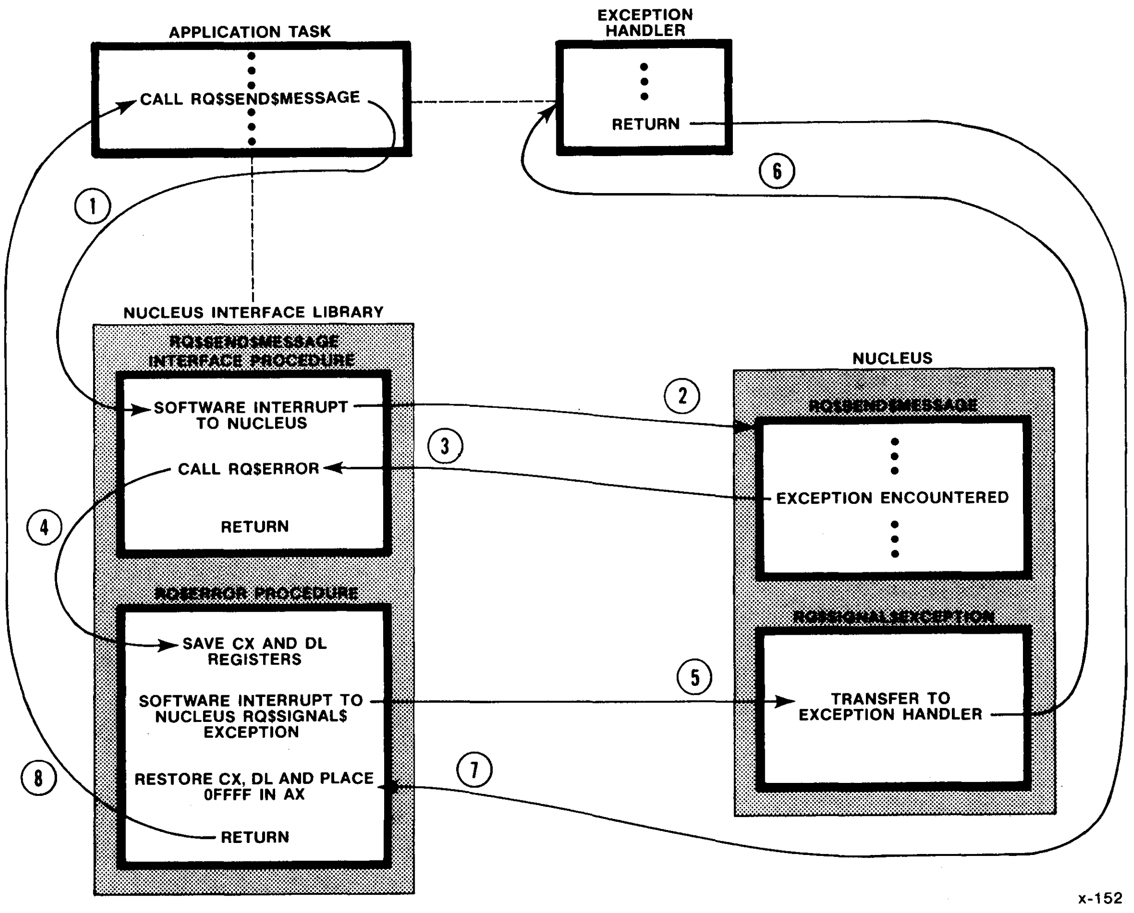

APPLICATION TASK
CALL RQ$SEND$MESSAGE
EXCEPTION HANDLER
RETURN
1
2
3
4
5
6
7
8
NUCLEUS INTERFACE LIBRARY
RQ$SEND$MESSAGE INTERFACE PROCEDURE
SOFTWARE INTERRUPT TO NUCLEUS
CALL RQ$ERROR
RETURN
RQ$ERROR PROCEDURE
SAVE CX AND DL REGISTERS
SOFTWARE INTERRUPT TO NUCLEUS RQ$SIGNAL$ EXCEPTION
RESTORE CX, DL AND PLACE 0FFFF IN AX
RETURN
NUCLEUS
RQ$SEND$MESSAGE
EXCEPTION ENCOUNTERED
RQ$SIGNAL$EXCEPTION
TRANSFER TO EXCEPTION HANDLER
x-152

Figure 10-4. Handling Exceptions With an Exception Handler

The iRMX 86 Operating System uses this mechanism for returning exceptions to give subsystems and OS extensions flexibility in handling their own exceptions. They obtain this flexibility because they know that whenever they incur an exceptional condition, a routine in an interface library to which they are linked will call RQ$ERROR to process the exception. If they want their exceptional conditions to be processed in a special manner, they can provide their own version of RQ$ERROR to handle this special processing. Thus each subsystem and OS extension can process exceptional conditions in its own way.

As the creator of an OS extension, you have the option of linking your OS extension to the default RQ$ERROR procedure or providing one of your own. If you have an exception handler associated with your OS extension, you will probably want to use the default RQ$ERROR procedure. You will also want to use SET$EXCEPTION$HANDLER and GET$EXCEPTION$HANDLER, as described previously, to ensure that your exception handler is actually called in the event of an exceptional condition.

However, if your OS extension does not have an exception handler, it should handle exceptions in-line, so that it can then return the proper exception code to the task (or OS extension) that invoked your customized system calls. You can provide this feature by linking your OS extension to a version of RQ$ERROR that does not call SIGNAL$EXCEPTION. Instead, this RQ$ERROR procedure should place OFFFFH in the AX register (so that OFFFFH is returned for system calls that are invoked as functions) and then do a RETURN, to return control directly to the interface library. The interface library then returns control to your OS extension, allowing the OS extension to process the exception in-line. Figure 10-5 illustrates the flow of control for an OS extension that processes its exceptions in-line. The RQ$ERROR procedure in Figure 10-5 simply sets AX and does a RETURN.

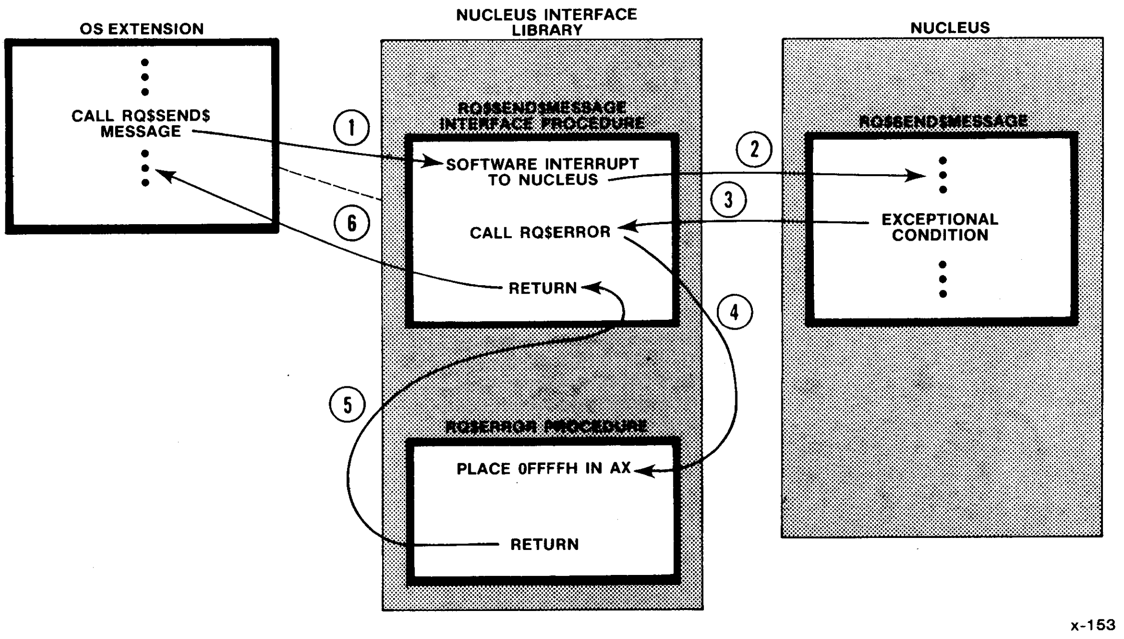


Figure 10-5. OS Extension Handling Exceptions In-Line

Even though your OS extension processes its own exceptions in-line, it will still want to return exceptions to tasks (or other OS extensions) that invoke the customized system calls. This involves having the entry (or function) procedure of your OS extension place the condition code and parameter number in CX and DL and then having the interface procedure call RQ$ERROR in the event of an exceptional condition. The "Interface Procedures" and "Entry Procedures" section of this chapter describe this procedure in detail. Because your OS extension returns the exception to the inteface procedure linked to the application task (or another OS extension), the RQ$ERROR procedure that gets called is the one in the interface library linked to the calling task, not the one in the interface library linked to the OS extension.

Figure 10-6 illustrates the flow of control for an OS extension that incurs an exceptional condition, processes the exception in-line, and then returns an exception to the application task that called it. Notice that both the OS extension and the application task, although not linked together, are each linked to interface libraries and an RQ$ERROR procedure. The RQ$ERROR procedure linked to the OS extension returns control back to the OS extension. The RQ$ERROR procedure linked to the application task is the default one; it calls SIGNAL$EXCEPTION.

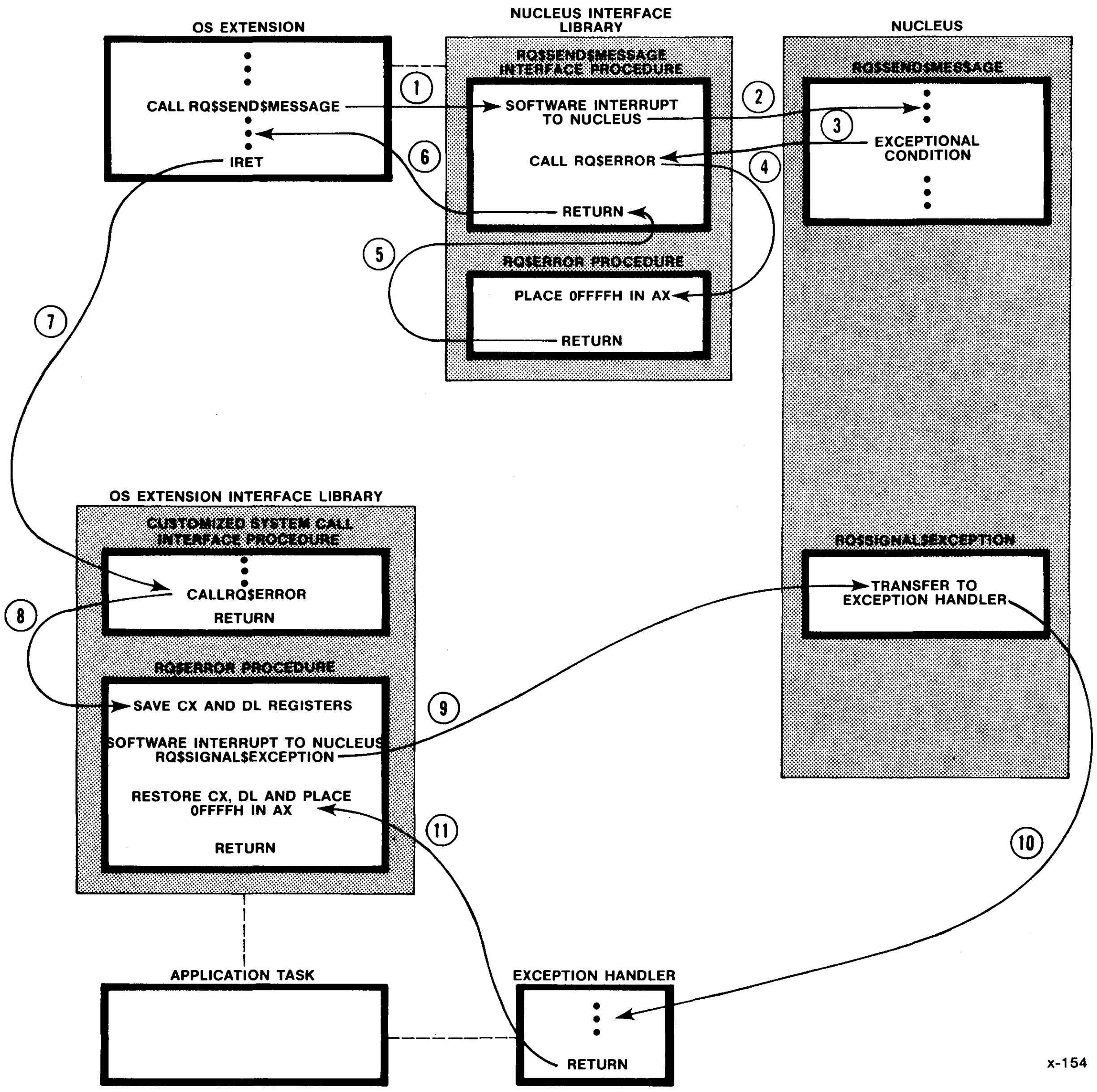


Figure 10-6. Control Flow for OS Extension and Application Task

Linking the Procedures

For each OS extension, you should produce several libraries of interface procedures. In fact, you should produce one library for each PL/M-86 model in which the calling task can be written. Within each library, you should have one interface procedure for each system call of the OS extension. Each job in your system should be linked to the appropriate interface library for each OS extension that the job calls.

For each OS extension, the entry procedure (if any) and the function procedures should all be linked together, along with any Operating System interface libraries that the procedures need. They should not be linked to any application code, since they are connected to the application tasks via the interrupt vector.

Any RQ$ERROR procedures that you create should be linked to the appropriate routines. If you create a special RQ$ERROR procedure that your interface procedures call whenever they inform the application task of an exception, you should place that RQ$ERROR procedure in the interface library you create. If you create an RQ$ERROR procedure to process exceptions that your OS extension incurs, you should link this RQ$ERROR procedure directly to the entry and function procedures. You should also link the Nucleus interface library, and the interface libraries for any of the other subsystems that you use, to both the application task and the OS extension. If you provide your own RQ$ERROR procedure, either for your interface procedures to call or to process exceptions in your OS extension, this procedure must precede the Nucleus interface library in the link sequence.

INITIALIZING THE INTERRUPT VECTOR

Before an interface procedure can successfully transfer control to an OS extension, the interrupt vector must be initialized with the addresses of the entry (or function) procedures. The SET$OS$EXTENSION system call is available for this purpose.

Because the interrupt vector must be initialized before any OS extensions are invoked, you must ensure that the initialization happens shortly after the system begins running. This can be accomplished during the initialization process described in the iRMX 86 CONFIGURATION GUIDE.

PROTECTING RESOURCES FROM BEING DELETED

Normally, an object can be deleted by a call to the deletion system call corresponding to the object's type. However, OS extensions can use the DISABLE$DELETION system call to make the object immune to this kind of deletion. A subsequent call to ENABLE$DELETION removes the immunity.

An object can have its deletion disabled more than once. Each call to DISABLE$DELETION must be countered by a call to ENABLE$DELETION before the object can be deleted. An object's disabling depth at any given moment is defined to be the number of times the object has had its deletion disabled minus the number of times its deletion has been enabled. Usually, an object cannot be deleted until its disabling depth is zero. The lone exception is that a call to FORCE$DELETE deletes objects whose disabling depth is one. Also, calling ENABLE$DELETION for an object whose deletion depth is zero results in the E$CONTEXT exception code.

All of these system calls--DISABLE$DELETION, ENABLE$DELETION, and FORCE$DELETE--should be used only by OS extensions.

NOTE

When a task attempts to delete an object whose disabling depth is too high to permit deletion, that task goes to sleep. The task remains asleep until the object's deletion depth becomes small enough to permit deletion. At that time, the object is deleted and the task is awakened. Because these circumstances can cause system deadlock, your tasks should exercise caution when deleting objects.

SYSTEM CALLS USED IN EXTENDING THE OPERATING SYSTEM

The following system calls are used extensively by OS extensions:

- DISABLE$DELETION

  This system call increases the deletion disabling depth of an object by one.

- ENABLE$DELETION

  This system call removes one level of deletion disabling from an object, reversing the effect of one DISABLE$DELETION call.

- FORCE$DELETE

  This system call deletes objects whose disabling depths are one or zero.

- SET$OS$EXTENSION

  This system call can be used either to place an address in a specific entry of the interrupt vector or to remove such an entry.

- SIGNAL$EXCEPTION

  This system call advises a task than an exceptional condition has occurred in an OS extension that the task has called.

CHAPTER 11. TYPE MANAGERS

The object types and system calls provided by the Nucleus and I/O System are sufficient for many applications. However, some applications have special requirements that would best be met if the iRMX 86 Operating System had additional object types and system calls for manipulating objects of those types. A type manager is an operating system extension that provides these services.

If your system requires additional object types, you must write a type manager for each of those types. The responsibilities of each type manager include:

- Implementing a new type by creating objects of the new type.
- Providing a mechanism for deleting objects of the new type.
- Optionally providing the system calls that application tasks can invoke to create, manipulate, and delete objects of the new type.

This chapter describes creating and deleting objects of new type. Chapter 10 describes extending the Operating System to include new system calls. An example appears at the end of this chapter which combines both of these operations.

CREATING NEW OBJECTS

Creating custom-made objects is a two-step process:

1. Create the type.
2. Create objects of that type.

The CREATE$EXTENSION system call creates the type. CREATE$EXTENSION accepts a new type code as a parameter and returns a token for the new type. The token represents a license to create objects of the new type.

The CREATE$COMPOSITE system call creates objects of the new type. CREATE$COMPOSITE accepts as a parameter the token returned from CREATE$EXTENSION. CREATE$COMPOSITE also accepts as input a list of tokens for the objects that are to compose the new object (the _component objects_) and returns a token for the new object, called a _composite object_.

Figure 11-1 illustrates the creation process for composite objects.

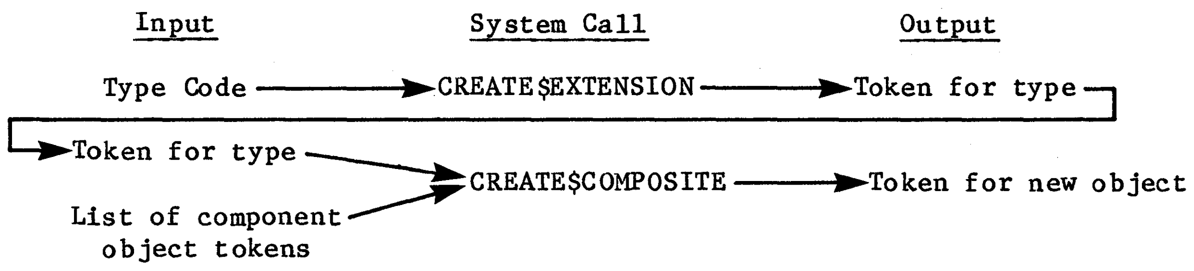


Figure 11-1. The Creation Sequence for Composite Objects

You should take note of two facts concerning the process of creating a composite object.

- First, its components, called component objects, are all iRMX 86 objects, either Intel- or user-provided.
- Second, no structure is imposed upon composite objects of a given extension type. Two objects of the same extension type can be, if desired, completely different in structure or in the number of components objects they comprise. This feature allows for maximum flexibility in the creation of new objects.

Once a type manager creates a new object type by calling CREATE$EXTENSION, that type manager owns the type. It is the only type manager that can create composite objects of that type. In addition, when it creates composite objects, the type manager can request that the composite object be sent back to the type manager when the object has to be deleted. Later sections describe this in detail.

MANIPULATING COMPOSITE OBJECTS AND EXTENSION TYPES

Two system calls are available for manipulating existing composite objects: INSPECT$COMPOSITE and ALTER$COMPOSITE. INSPECT$COMPOSITE returns a list of component tokens for a composite object. ALTER$COMPOSITE replaces a token in the component list of a composite object, either with another token or with a null.

DELETING COMPOSITE OBJECTS AND EXTENSION TYPES

Two system calls are available exclusively for deleting composite objects: DELETE$COMPOSITE and DELETE$EXTENSION. DELETE$COMPOSITE deletes a particular composite object (but not its components); DELETE$EXTENSION deletes a specified extension type and either deletes the composites of that type or sends them to a deletion mailbox, in which case the type manager must delete them.

A third system call, DELETE$JOB, also deletes composite objects as a part of its processing. Although DELETE$JOB cannot delete extension types (in fact, DELETE$JOB returns an exception code if the job contains any extension objects), it can delete composites or send them to deletion mailboxes where the type managers for these objects must delete them.

The deletion$mailbox parameter in the CREATE$EXTENSION system call determines whether DELETE$EXTENSION and DELETE$JOB actually delete composite objects or instead send them to deletion mailboxes. There are two possibilities for this option.

If you specify a zero for the deletion$mailbox parameter of CREATE$EXTENSION, then DELETE$EXTENSION and DELETE$JOB assume all responsibility for deleting extension and composite objects. Your type manager plays no part in the deletion process and you can skip the next three sections of this chapter.

However, if you specify a token for a mailbox in the deletion$mailbox parameter of CREATE$EXTENSION, then DELETE$EXTENSION and DELETE$JOB send tokens for all composite objects of the indicated type to the mailbox instead of actually deleting these objects. Your type manager for that extension type is then responsible for deleting the composite objects.

There are two conditions that must occur before the type manager receives tokens for composite objects via the previously mentioned deletion mailbox:

- Your type manager, when it called CREATE$EXTENSION, must have filled in the deletion$mailbox parameter with a token for a mailbox.

- A task must call DELETE$EXTENSION or DELETE$JOB.

If these two conditions are met, the type manager is responsible for deleting the composite objects sent to the mailbox. The following sections describe the type manager's responsibilities in more detail.

TYPE MANAGER RESPONSIBILITIES DURING DELETE$JOB

When a task calls DELETE$JOB, the Nucleus normally deletes every object in the job. However, if the job contains a composite object whose extension has a deletion mailbox, the Nucleus sends the token for the composite object to the deletion mailbox. The Nucleus then waits until the type manager calls DELETE$COMPOSITE before continuing the deletion process.

The type manager has the following responsibilities for servicing the deletion mailbox.

1. First, it must wait at the deletion mailbox to receive the tokens for the objects to be deleted.

2. Next, it must perform any special processing that is required in order to delete the composite object. For example, it might want to wait until all tasks have stopped using the composite.

3. Then, it has the option of deleting those component objects that are not contained in the job being deleted. It cannot, however, delete objects contained in the job being deleted or it will incur an exceptional condition. This is not a problem because the objects that are a part of the job being deleted will automatically be deleted as part of the DELETE$JOB call.

4. Finally, it should call DELETE$COMPOSITE. This serves two purposes. It deletes the composite object (but not the component objects), and it informs the Nucleus that the type manager has finished the special processing neeeded to delete the composite object. After the type manager calls DELETE$COMPOSITE, the Nucleus resumes the DELETE$JOB processing.

The type manager must call DELETE$COMPOSITE each time the Nucleus sends a token for a composite object to the deletion mailbox because DELETE$COMPOSITE serves to return control back to the Nucleus. If the type manager fails to call DELETE$COMPOSITE, the DELETE$JOB system call will not finish processing. Figure 11-2 illustrates the type manager's involvement in the DELETE$JOB process.

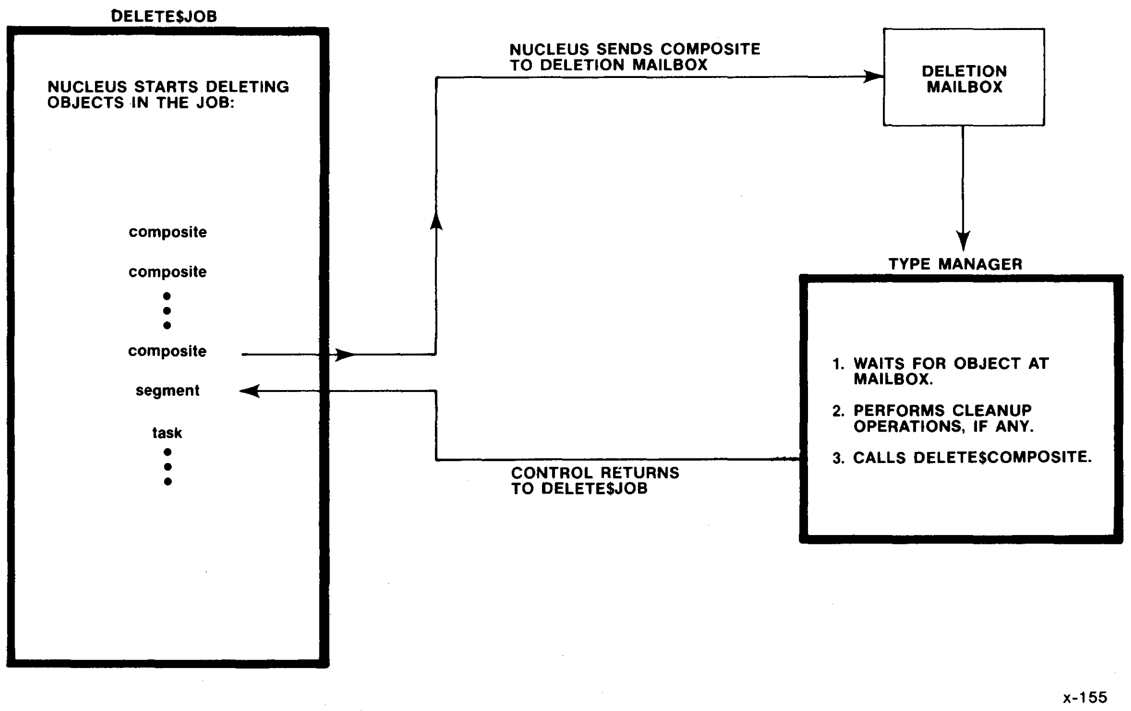

DELETE$JOB
NUCLEUS STARTS DELETING OBJECTS IN THE JOB:
composite
composite
composite
segment
task
NUCLEUS SENDS COMPOSITE TO DELETION MAILBOX
DELETION MAILBOX
TYPE MANAGER
1. WAITS FOR OBJECT AT MAILBOX.
2. PERFORMS CLEANUP OPERATIONS, IF ANY.
3. CALLS DELETE$COMPOSITE.
CONTROL RETURNS TO DELETE$JOB
x-155

Figure 11-2. Type Manager Involvement in DELETE$JOB

Note that the type manager is not required to delete all component objects. In the course of DELETE$JOB, the Nucleus deletes any Nucleus objects in the job. The Nucleus sends stopstartstop

TYPE MANAGER RESPONSIBILITIES DURING DELETE$EXTENSION

A task can call DELETE$EXTENSION to delete an extension type. This is useful when the license to create composite objects of a given extension type is no longer needed. When a task calls DELETE$EXTENSION and the extension has a deletion mailbox, the Nucleus sends the tokens for all composite objects of that extension type to the deletion mailbox. After sending a token for an object to the deletion mailbox, the Nucleus waits until the type manager calls DELETE$COMPOSITE before sending the next composite.

The type manager has similar responsibilities during DELETE$EXTENSION that it has during DELETE$JOB. First it must wait at the deletion mailbox for the objects' tokens. Then it must handle any special processing necessary to delete the object. Finally it must call DELETE$COMPOSITE to delete the composite. As with DELETE$JOB, the type manager must call DELETE$COMPOSITE for each token it receives at the deletion mailbox. If it does not do this, the DELETE$EXTENSION system call will not finish processing.

However, unlike the situation during DELETE$JOB, the type manager has the choice during DELETE$EXTENSION of whether or not to delete individual component objects. If it wishes the component objects to be deleted, the type manager must explicitly delete these objects. Unlike DELETE$JOB, the DELETE$EXTENSION system call does not automatically delete component objects.

DELETION OF NESTED COMPOSITES

Since a composite object can contain objects of any kind, it is possible for some of its component objects to be composite objects themselves. This situation can cause problems for type managers when they delete the composite objects if the type manager for any of the composite objects depends on the existence of any of the other composite objects in order to complete its processing.

For example, suppose objects A and B are composites of different extension types and B is a component of A. Each of the composites has a type manager that performs special cleanup functions before it can delete the corresponding composite. If neither of the type managers requires information contained in the other composite in order to perform its special processing, the deletion process can proceed without difficulty.

However, if for some reason the type manager for composite A requires some information contained in composite B in order to complete its processing, the deletion process becomes more complex. In order for this deletion scheme to work, you must guarantee that composite A will be deleted before composite B. This requires that you know the order in which the Nucleus deletes objects and sends composites to deletion mailboxes, so that you can set up your composites correctly.

The Nucleus deletes composite objects before it deletes non-composite objects. It deletes composite objects on a last-in/first-out basis; that is, in the reverse order from which they were created. Therefore, a type manager can depend on receiving the tokens for composite objects that it creates before the Nucleus deletes the component objects contained in them. The only exception to this is when a composite (composite A) is created before another composite (composite B) and composite B is inserted as a component into composite A using the ALTER$COMPOSITE system call. In this case, composite B will be deleted first, and the type manager of composite A cannot rely on the existence of composite B when it receives composite A for deletion.

WRITING A TYPE MANAGER

A type manager consists of two parts:

- The initialization part creates the type and optionally creates a deletion mailbox to which the system can send tokens for objects of the type when deleting either jobs or the type itself.
- The service part provides the system calls through which tasks can create and manipulate objects of the type.

Because the initialization phase must be completed before any task attempts to obtain tokens for objects, the initialization part should be written as a task that executes early in the life of the system. To ensure early execution, the task should be part of the initialization task of a first-level user job in the job tree. Refer to the iRMX 86 CONFIGURATION GUIDE for information concerning first-level jobs.

The service part of the type manager is written as an operating system extension. Refer to Chapter 10 for information about operating system extensions.

The best way to learn about type managers is to study an example. The following example presents the main parts of a type manager for ring buffers.

EXAMPLE -- A RING BUFFER MANAGER

This example shows the most educational portions of a ring buffer manager. It also serves to illustrate the various parts of an operating system extension. Be advised, however, that the example is incomplete and therefore should be imitated only with discretion. In particular, the example has the following shortcomings:

- The issue of exception handling is not addressed. Clearly the code supporting a system call should examine each invocation for validity, but, for brevity, the ring buffer example does not do this.
- There are no safeguards against partial creation of an object. When creating a composite object, a type manager must first create the components of the object. Occasionally, after creating some of the components, the manager might be unable to create the others. A type manager should be able to recover from this situation, usually by deleting the components already created and returning an exception code to the caller. The example, again for brevity, does not do this.
- The entry routine does not check the entry code for validity.
- The potential for problems with deletion is ignored. For this reason, you should imagine that the environment of the example is constrained in at least two ways. First, only one task will ever try to delete a ring buffer and, when it does try, no other task will be using that buffer. Second, when a job containing a task that created a ring buffer is deleted, no tasks in other jobs are using that ring buffer.
- The example has been desk-checked, but the example has not actually been tested.

A ring buffer is a block of memory in which bytes of data are placed at successively higher addresses. Interspersed with byte insertions are byte removals, with the restriction that the byte being removed must always be the byte that has been in the buffer for the longest time. Thus, data enters and leaves a ring buffer in a first-in-first-out manner. Ring buffers get their name from the fact that the lowest address logically follows the highest address. That is, if the last byte placed in (or retrieved from) the buffer is at its highest address, then the next byte to be placed in it (or retrieved from it) is at the lowest address. As data enters and leaves the buffer, the portion contaning data "runs" around the ring, with the pointer to the last byte out "chasing" the pointer to the last byte in. Figure 11-3 illustrates these characteristics.

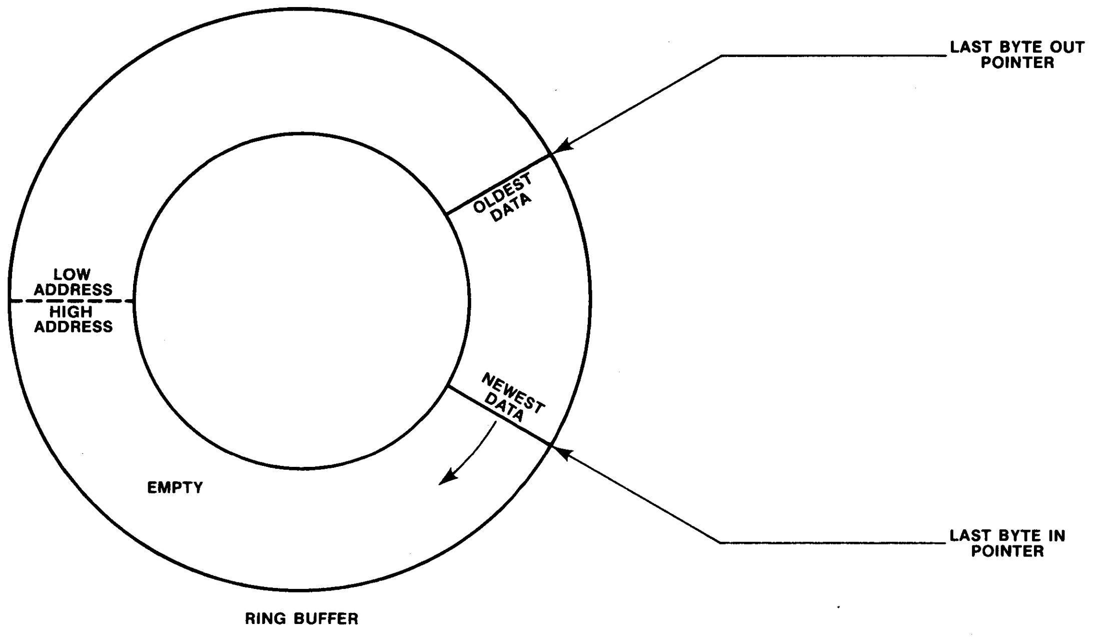


Figure 11-3. A Ring Buffer

The main (service) part of the example consists of four procedures: CREATE_RING_BUFFER, DELETE_RING_BUFFER, PUT_BYTE, and GET_BYTE. The last two procedures are for placing a character in a ring buffer, and for retrieving a character, respectively.

```
/***********************************************************************
 *  NOTE:  The following common literal file (COMMON.LIT) is included   *
 *  in each of the PL/M 86 portions of the example.                     *
 ***********************************************************************/
   DECLARE TOKEN        LITERALLY 'SELECTOR';
                               /* if your PL/M compiler does
                                  not support this variable
                                  type, declare TOKEN a WORD */
```

```
DECLARE forever        LITERALLY 'WHILE 1';
DECLARE indefinitely   LITERALLY 'OFFFFH';
DECLARE ASTR$STRUC     LITERALLY 'STRUCTURE(
                                num$slots        WORD,
                                num$components   WORD,
                                seg              TOKEN,
                                empty$ct         TOKEN,
                                full$ct          TOKEN)';
DECLARE POINTER$STRUC  LITERALLY 'STRUCTURE(
                                offset           WORD,
                                base             WORD)';
DECLARE SEGMENT$STRUC  LITERALLY 'STRUCTURE(
                                size             WORD,
                                head             WORD,
                                tail             WORD,
                                buffer(1)        BYTE)';
```

THE INITIALIZATION PART

The initialization part creates a region to protect data in ring buffers from being manipulated by more than one task at a time. This part also creates the required extension type, creates a deletion mailbox, sets the operating system extension at entry 224 of the interrupt vector table, and then waits at the deletion mailbox. Code for the initialization part includes the following:

```
$INCLUDE(:Fx:COMMON.LIT);        /* Declares common literals */

RINGBUFFERMANAGER:  PROCEDURE EXTERNAL;
END RINGBUFFERMANAGER;

DECLARE ring$buffer$type         TOKEN PUBLIC;
DECLARE ring$buffer$region       TOKEN PUBLIC;

  RING_BUFFER_INIT:
   PROCEDURE;
   DECLARE delete$object         TOKEN;
   DECLARE exception             WORD;
   DECLARE fifo                  LITERALLY '0';
   DECLARE rb$code               LITERALLY '8000H';
   DECLARE deletion$mbox         TOKEN;
   DECLARE response$mbox         TOKEN;

   ring$buffer$region = RQ$CREATE$REGION    (fifo,
                                             @exception);
   deletion$mbox = RQ$CREATE$MAILBOX        (fifo,
                                             @exception);
   ring$buffer$type = RQ$CREATE$EXTENSION   (rb$code,
                                             deletion$mbox,
                                             @exception);
```

```
    CALL RQ$SET$OS$EXTENSION                  (224,
                                              @ring$buffer$manager,
                                              @exception);

    CALL RQ$END$INIT$TASK;
    DO FOREVER;
      delete$object = RQ$RECEIVE$MESSAGE     (deletion$mbox,
                                              indefinitely,
                                              @response$mbox,
                                              @exception);
      CALL RQ$DELETE$COMPOSITE               (ring$buffer$type,
                                              delete$object,
                                              @exception);

/***********************************************************************
 * If desired, delete the components of the composite object.  They are *
 * not automatically deleted when DELETE$EXTENSION is called.  See the  *
 * DELETE$RING$BUFFER procedure, shown later, for the code that does    *
 * this.                                                                *
 ***********************************************************************/

    END RING_BUFFER_INIT;
```

The variable ring$buffer$manager is a pointer to the entry procedure of the operating system extension.

THE INTERFACE LIBRARY

The user interface library consists of four small procedures, one for each of the system calls provided by the operating system extension. The library supports application code written in the PL/M-86 "large" model. If a different model had been used for compiling the application code, these interface procedures would be slightly different, reflecting the fact that, when making procedure calls in other models, the stack is used differently than in the large model. The interface procedures are as follows:

```
  CREATERB  PROC    FAR
            PUBLIC  RQCREATERB
            PUSH    BP          ;Save the BP value
            MOV     BP,SP
            LEA     SI,SS: BP+6 ;SS:SI contains location
                                ;  of first parameter
            MOV     BX,0        ;Code for CREATE_RING_BUFFER
            INT     224         ;Call the extension
            POP     BP          ;Restore the BP value
            RET     2           ;Passing one argument
  CREATERB  ENDP


  DELETERB  PROC    FAR
            PUBLIC  RQDELETERB
            PUSH    BP
            MOV     BP,SP
```

```
           LEA     SI,SS: BP+6
           MOV     BX,1          ;Code for DELETE_RING_BUFFER
           INT     224
           POP     BP
           RET     2             ;Passing one argument
DELETERB   ENDP


GETRBBYTE PROC     FAR
          PUBLIC   RQGETBYTE
          PUSH     BP
          MOV      BP,SP
          LEA      SI,SS: BP+6
          MOV      BX,2          ;Code for GET_BYTE
          INT      224
          POP      BP
          RET      2             ;Passing one argument
GETRBBYTE ENDP


PUTRBBYTE PROC     FAR
          PUBLIC   RQPUTBYTE
          PUSH     BP
          MOV      BP,SP
          LEA      SI,SS: BP+6
          MOV      BX,3          ;Code for PUT_BYTE
          INT      224
          POP      BP
          RET      4             ;Passing two arguments
PUTRBBYTE ENDP
```

These interface procedures correspond to a set of external procedure declarations in the application PL/M-86 code:

```
CREATERB:  PROCEDURE(size)     WORD EXTERNAL;
  DECLARE size                 WORD;
END CREATERB;


DELETERB:  PROCEDURE(ring$buffer$token) EXTERNAL;
  DECLARE ring$buffer$token    TOKEN;
END DELETERB;


GETRBBYTE:  PROCEDURE(ring$buffer$token) BYTE EXTERNAL;
  DECLARE ring$buffer$token    TOKEN;
END GETRBBYTE;


PUTRBBYTE:  PROCEDURE(char, ring$buffer$token) EXTERNAL;
  DECLARE char                 BYTE;
  DECLARE ring$buffer$token    WORD;
END PUTRBBYTE;
```

THE ENTRY PROCEDURE

The entry procedure in the operating system extension is as follows:

```
                     PUBLIC RINGBUFFERMANAGER
                     EXTRN  CREATERINGBUFFER:FAR
                     EXTRN  DELETERINGBUFFER:FAR
                     EXTRN  GETBYTE:FAR
                     EXTRN  PUTBYTE:FAR
  FLAGS              EQU     BP+8
  RINGBUFFERMANAGER  PROC   FAR
                     PUSH   DS       ;Push user values not
                     PUSH   BP       ;  automatically saved
                     MOV    BP,SP    ;Value of BP equals
                                     ;  stackpointer and is
                                     ;  used in any calls
                                     ;  from this operating
                                     ;  system extension to
                                     ;  SIGNAL$EXCEPTION
                     PUSH   FLAGS        ;Restore
                     POPF                ;  saved flags
                     PUSH   SS           ;Base of pointer to
                                         ;  parameters
                     PUSH   SI           ;Offset of pointer
                                         ;  to parameters
                     SHL    BX,1         ;Call the appropriate
                     SHL    BX,1         ;  extension
                     CALL   CS:TABLE BX  ;  procedure
                     POP    BP           ;Restore saved BP
                     POP    DS           ;  and DS values
                     IRET
  TABLE              DD  CREATERINGBUFFER;  The addresses
                     DD  DELETERINGBUFFER;   of the utility
                     DD  GETBYTE         ;   procedures in
                     DD  PUTBYTE         ;   the OS extension
  RINGBUFFERMANAGER  ENDP
```

Note that the entry routine is completely independent of the PL/M-86 model used when compiling the application code. The interface library conceals the choice of model from the entry procedure.

THE CREATE_RING_BUFFER PROCEDURE

The sole function of the CREATE_RING_BUFFER procedure is to create a ring buffer for the calling task and to return to the task a token for the composite ring buffer object.

Each ring buffer consists of three objects: a segment and two semaphores. The supporting data structure, required by the iRMX 86 Operating System for calls to CREATE$COMPOSITE and INSPECT$COMPOSITE, has the following five fields:

- The number of slots available for tokens in the following list of component object tokens. Because ring buffers are composed of three objects and there is no apparent reason to add components at a later time, the number of slots is set to three.

- The number of component objects actually in the composite object. In this case, the number of components is three.

- A token for a segment. The segment contains the ring buffer. The first word in the segment contains the size of the actual ring buffer. The second word of the segment is a "pointer" to the most recently entered byte in the buffer, while the third word points to the oldest byte in the buffer. The remainder of the segment is to be used as the buffer itself. Note that, in the program, a structure reflecting the intended breakdown of the segment is superimposed on the segment.

- A token for a semaphore. This semaphore is used to keep track of the number of vacancies in the ring buffer. Thus, it is initialized to the size of the buffer.

- A token for a semaphore. This semaphore is used to keep track of the number of occupied bytes in the ring buffer. Thus, it is initialized to zero.

The CREATE_RING_BUFFER routine creates the components of the composite ring buffer object, initializes the appropriate fields, and then creates the composite object, as follows:

```
    $INCLUDE(:Fx:COMMON.LIT);        /* Declares common literals */
    DECLARE ring$buffer$type         TOKEN EXTERNAL;

  CREATE_RING_BUFFER:
    PROCEDURE (param$ptr) TOKEN PUBLIC REENTRANT;
    DECLARE param$ptr                POINTER;
    DECLARE size BASED param$ptr     WORD;
    DECLARE seg$ptr                  POINTER;
    DECLARE ptr$struc                POINTER$STRUC AT (@seg$ptr);
    DECLARE astr                     ASTR$STRUC;
    DECLARE segment                  SEGMENT$STRUC BASED seg$ptr;
    DECLARE exception                WORD;
    DECLARE ring$buffer              TOKEN;
    DECLARE priority                 LITERALLY '1';
    astr.num$slots = 3;
    astr.num$components = 3;
    astr.seg = RQ$CREATE$SEGMENT          (size+6,
                                          @exception);
    astr.empty$ct = RQ$CREATE$SEMAPHORE   (size, size,
                                          priority,
                                          @exception);
    astr.full$ct = RQ$CREATE$SEMAPHORE    (0,
                                          size,
                                          priority,
                                          @exception);
    ptr$struc.base = astr.seg;
    ptr$struc.offset = 0;
    segment.size = size;
    segment.head = -1;
    segment.tail = 0;
    ring$buffer = RQ$CREATE$COMPOSITE     (ring$buffer$type,
                                          @astr,
                                          @exception);
    RETURN ring$buffer;
    END CREATE_RING_BUFFER;
```

The segment.head variable is set to -1 because the PUT_BYTE procedure (shown later) advances this pointer <u>before</u> placing a character in the buffer.

THE DELETE_RING_BUFFER PROCEDURE

DELETE_RING_BUFFER can be called by any task wanting to delete a ring buffer:

```
    $INCLUDE(:Fx:COMMON.LIT);    /* Declares common literals */
    DECLARE ring$buffer$type     TOKEN EXTERNAL;

  DELETE_RING_BUFFER:
    PROCEDURE(param$ptr) REENTRANT PUBLIC;
    DECLARE param$ptr            POINTER;
    DECLARE ring$buffer$token BASED param$ptr TOKEN;
    DECLARE astr                 ASTR$STRUC;
    DECLARE exception            WORD;

    astr.num$slots = 3;
    CALL RQ$INSPECT$COMPOSITE    (ring$buffer$type,
                                 ring$buffer$token,
                                 @astr,
                                 @exception);
    CALL RQ$DELETE$COMPOSITE     (ring$buffer$type,
                                 ring$buffer$token,
                                 @exception);
    CALL RQ$DELETE$SEGMENT       (astr.seg,
                                 @exception);
    CALL RQ$DELETE$SEMAPHORE     (astr.empty$ct,
                                 @exception);
    CALL RQ$DELETE$SEMAPHORE     (astr.full$ct,
                                 @exception);
    END DELETE_RING_BUFFER;
```

THE PUT_BYTE PROCEDURE

The PUT_BYTE procedure places a character in the buffer by advancing the "pointer" to the front of the buffer and then placing the character in the byte being pointed to:

```
    $INCLUDE(:Fx:COMMON.LIT);    /* Declares common literals */
    DECLARE ring$buffer$type     TOKEN EXTERNAL;
    DECLARE ring$buffer$region   TOKEN EXTERNAL;

  PUT_BYTE:
    PROCEDURE(param$ptr) REENTRANT PUBLIC;
    DECLARE param$ptr            POINTER;
    DECLARE params BASED param$ptr STRUCTURE(
            ring$buffer$token    TOKEN,
            char                 BYTE);
    DECLARE size                 WORD;
    DECLARE seg$ptr              POINTER;
    DECLARE ptr$struc            POINTER$STRUC AT (@seg$ptr);
    DECLARE astr                 ASTR$STRUC;
    DECLARE segment              SEGMENT$STRUC BASED seg$ptr;
    DECLARE exception            WORD;
    DECLARE units$left           WORD;

    astr.num$slots = 3;
    CALL RQ$INSPECT$COMPOSITE        (ring$buffer$type,
                                     params.ring$buffer$token,
                                     @astr,
                                     @exception);
    units$left = RQ$RECEIVE$UNITS    (astr.empty$ct,
                                     1,
                                     indefinitely,
                                     @exception);
    CALL RQ$RECEIVE$CONTROL          (ring$buffer$region,
                                     @exception);
    ptr$struc.base = astr.seg;
    ptr$struc.offset = 0;
    segment.head = ((segment.head + 1) MOD segment.size);
    segment.buffer(segment.head) = params.char;
    CALL RQ$SEND$CONTROL             (@exception);
    CALL RQ$SEND$UNITS               (astr.full$ct,
                                     1,
                                     @exception);
    END PUT_BYTE;
```

Note that this procedure enters a region after obtaining the desired unit. To reverse these steps would create a deadlock situation, particularly if the same reversal occurs in the GET_BYTE routine (shown later).

Note also that the order of the parameters ring$buffer$token and char is the opposite of the order of those parameters in the earlier external declaration of PUTRBBYTE. This is typical of procedures with multiple arguments and results from the use of the stack when passing parameters.

THE GET_BYTE PROCEDURE

GET_BYTE removes the oldest byte in the buffer and then advances the segment.tail "pointer":

```
    $INCLUDE(:Fx:COMMON.LIT);     /* Declares common literals */
    DECLARE ring$buffer$type        TOKEN EXTERNAL;
    DECLARE ring$buffer$region      TOKEN EXTERNAL;

  GET_BYTE: PROCEDURE(param$ptr) BYTE PUBLIC REENTRANT;
    DECLARE param$ptr POINTER;
    DECLARE ring$buffer$token BASED param$ptr TOKEN;
    DECLARE size                    WORD;
    DECLARE seg$ptr                 POINTER;
    DECLARE ptr$struc               POINTER$STRUC AT (@seg$ptr);
    DECLARE astr                    ASTR$STRUC;
    DECLARE segment                 SEGMENT$STRUC BASED seg$ptr;
    DECLARE exception               WORD;
    DECLARE char                    BYTE;
    DECLARE units$left              WORD;

    astr.num$slots = 3;
    CALL RQ$INSPECT$COMPOSITE       (ring$buffer$type,
                                    ring$buffer$token,
                                    @astr,
                                    @exception);
    units$left = RQ$RECEIVE$UNITS   (astr.full$ct,
                                    1,
                                    indefinitely,
                                    @exception);
    CALL RQ$RECEIVE$CONTROL         (ring$buffer$region,
                                    @exception);
    ptr$struc.base = astr.seg;
    ptr$struc.offset = 0;
    char = segment.buffer(segment.tail);
    segment.tail = ((segment.tail + 1) MOD segment.size);
    CALL RQ$SEND$CONTROL            (@exception);
    CALL RQ$SEND$UNITS              (astr.e,pty$ct,
                                    1,
                                    @exception);
    RETURN char;
    END GET_BYTE;
```

EPILOGUE

This completes the important parts of the example (recall that the example is not complete). Any task in any job linked to these procedures may call any one of the procedures. The procedure names to be used in such calls are CREATE$RB, DELETE$RB, GET$RB$BYTE, and PUT$RB$BYTE. Note that application programs cannot manipulate either ring buffers or their component objects, except through these system calls. In fact, there is no need for application programmers to be aware that ring buffers are composed of several other objects. To them, ring buffers appear (except for the absence of 'RQ' in the procedure names) to be standard iRMX 86 objects.

SYSTEM CALLS FOR TYPE MANAGERS

The following system calls enable type managers to manipulate extension and composite objects:

- ALTER$COMPOSITE

  This system call replaces a component in a composite object with either a null or another object.

- CREATE$COMPOSITE

  This system call creates a composite object of a specified extension type.

- CREATE$EXTENSION

  This system call creates an extension object which may subsequently be used as a license for creating composite objects. Input includes a token for a mailbox where these composite objects are sent for deletion.

- DELETE$COMPOSITE

  This system call deletes a composite object.

- DELETE$EXTENSION

  This system call deletes an extension object and sends all composite objects of that extension type to the associated deletion mailbox.

- INSPECT$COMPOSITE

  This system call returns a list of the component object tokens contained in a composite object.

CHAPTER 12. NUCLEUS SYSTEM CALLS

This chapter contains the calling sequences and other information about the system calls to the Nucleus. The system calls are listed in alphabetical order. Names of the calls are written in white on a dark background in the upper outside corner of each page. The calling sequence for each call is that for the PL/M-86 interface. The information for each system call is organized into the following categories, in the following order:

- A brief sketch of the effects of the call.
- The PL/M-86 calling sequence for the system call.
- Definitions of the input parameters, if any.
- Definitions of the output parameters, if any.
- A detailed description of the effects of the call.
- An example of how the system call can be used.
- The condition codes that can result from using the call, with a description of the possible causes of each condition.

PL/M-86 data types, such as BYTE, WORD, and SELECTOR, are used throughout the chapter. They are always capitalized and their definitions are found in Appendix A. Also, the iRMX 86 data type TOKEN is capitalized throughout the chapter. If your compiler supports the SELECTOR data type, a TOKEN can be declared literally either SELECTOR or WORD. The word "token" in lower case refers to a value that the iRMX 86 Operating System assigns to an object. The Operating System returns this value to a TOKEN (the data type) when it creates the object.

The examples used in this chapter assume the reader is familiar with PL/M-86. In these examples, the appropriate DECLARE statements are made first. Before the first of these DECLARE statements is an INCLUDE statement that declares all of the system calls included in the iRMX 86 Operating System. Refer to the iRMX 86 PROGRAMMING TECHNIQUES manual for additional information on creating this INCLUDE statement. For the sake of simplicity, the examples assume that an established exception handler is to deal with exceptional conditions. Consequently, they do not illustrate in-line exception processing.

Between this introduction and the details of the system calls is a command dictionary, in which the calls are grouped according to type. This dictionary, which includes short descriptions and page numbers of the complete descriptions in this chapter, is provided as an alternate way of indexing the system calls.

COMMAND DICTIONARY

COMMAND DICTIONARY (continued)

COMMAND DICTIONARY (continued)

COMMAND DICTIONARY (continued)

COMMAND DICTIONARY (continued)

ACCEPT$CONTROL

The ACCEPT$CONTROL system call requests immediate access to data protected by a region.

CAUTION

Tasks which use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal) while the task is in the region.

```
CALL RQ$ACCEPT$CONTROL (region, except$ptr);
```

INPUT PARAMETER

region	A TOKEN for the target region.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The ACCEPT$CONTROL system call provides access to data protected by a region if access is immediately available. If access is not immediately available, the E$BUSY condition code is returned and the calling task remains ready.

EXAMPLE

```
/*************************************************************************
 *  This example illustrates how the ACCEPT$CONTROL system call can be  *
 *  used to access data protected by a region.                           *
 *************************************************************************/
   $INCLUDE(:F1:SAMPLE.EXT);          /* declares all system calls */

   DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                      /* if your PL/M compiler does not
                                         support this variable type,
                                         declare TOKEN a WORD */
   DECLARE region$token               TOKEN;
   DECLARE priority$queue             LITERALLY '1'; /* tasks wait in
                                                        priority order */
   DECLARE status                     WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;
          .
          .  Typical PL/M-86 Statements
          .

/*************************************************************************
 *  In order to access the data within a region, a task must know the   *
 *  token for that region.  In this example, the needed token is known  *
 *  because the calling task creates the region.                         *
 *************************************************************************/
   region$token = RQ$CREATE$REGION    (priority$queue,
                                      @status);
          .
          .  Typical PL/M-86 Statements
          .

/*************************************************************************
 *  At some point in the task, access is needed to the data protected   *
 *  by the region.  The calling task then invokes the ACCEPT$CONTROL    *
 *  system call and obtains access to the data if access is             *
 *  immediately available.                                               *
 *************************************************************************/
   CALL RQ$ACCEPT$CONTROL             (region$token,
                                      @status);
          .
          .  Typical PL/M-86 Statements
          .

/*************************************************************************
 *  When the task is ready to relinquish access to the data protected   *
 *  by the region, it invokes the SEND$CONTROL system call.             *
 *************************************************************************/
   CALL RQ$SEND$CONTROL               (@status);
          .
          .  Typical PL/M-86 Statements
          .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$BUSY	Another task currently has access to the data.
E$EXIST	The region parameter does not refer to a currently existing object.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	The region parameter does not contain a token for a region.

ALTER$COMPOSITE

The ALTER$COMPOSITE system call replaces components of composite objects.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal).

```
CALL RQ$ALTER$COMPOSITE(extension, composite, component$index,
                        replacing$obj, except$ptr);
```

INPUT PARAMETERS

extension	A TOKEN for the extension type object corresponding to the composite object being altered.
composite	A TOKEN for the composite object being altered.
component$index	A WORD whose value specifies the location (starting at 1) in the component list of the component to be replaced.
replacing$obj	A TOKEN for the replacement component object or zero, which represents no object.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The ALTER$COMPOSITE system call changes a component of a composite object. Any component in a composite object can be replaced either with a token for another object or with a place-holding zero that represents no object.

The component$index indicates the position of the target token in the list of components. A component$index value of three indicates the third component object in the list.

EXAMPLE

See the example in section "The GET_BYTE Procedure" of Chapter 11.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The composite parameter is not compatible with the extension parameter.
E$EXIST	One or both of the extension or composite parameters does not refer to a currently existing object.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$PARAM	The component$index parameter refers to a nonexistent position in the component object list.
E$TYPE	One or both of the extension or composite parameters is not of the correct object type.

CATALOG$OBJECT

CATALOG$OBJECT places an entry for an object in an object directory.

```
CALL RQ$CATALOG$OBJECT (job, object, name, except$ptr);
```

INPUT PARAMETERS

job	A TOKEN that indicates where the object is to be cataloged. • if zero, indicates that the object is to be cataloged in the object directory of the job to which the calling task belongs. • if not zero, the TOKEN for the job in whose object directory the object is to be cataloged.
object	A TOKEN for the object to be cataloged. A zero for this parameter indicates that a null token is being cataloged.
name	A POINTER to a STRING containing the name under which the object is to be cataloged. The name itself must not exceed 12 characters in length. Each character can be a byte consisting of any value from 0 to OFFH.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CATALOG$OBJECT system call places an entry for an object in the object directory of a specific job. The entry consists of both a name and a token for the object. There may be several such entries for a single object in a directory, because the object may have several names. (However, in a given object directory, only one object may be cataloged under a given name.) If another task is waiting, via the LOOKUP$OBJECT system call, for the object to be cataloged, that task is awakened when the entry is cataloged.

EXAMPLE

```
/*************************************************************************
 *  This example illustrates how the CATALOG$OBJECT system call can be  *
 *  used to place an entry in an object directory.                       *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                   /* if your PL/M compiler does not
                                      support this variable type,
                                      declare TOKEN a WORD */
    DECLARE mbx$token              TOKEN;
    DECLARE mbx$flags              WORD;
    DECLARE job                    TOKEN;
    DECLARE status                 WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    mbx$flags = 0;                 /* designates four objects to be queued
                                      on the high performance object
                                      queue;  designates a first-in/
                                      first-out task queue. */

    job = 0;                       /* indicates objects to be cataloged
                                      into the object directory of the
                                      calling task's job */
        .
        .  }    Typical PL/M-86 Statements
        .

/*************************************************************************
 *  The calling task creates an object, in this example a mailbox,      *
 *  before cataloging the object's token.                                *
 *************************************************************************/
    mbx$token = RQ$CREATE$MAILBOX (mbx$flags,
                                   @status);
        .
        .  }    Typical PL/M-86 Statements
        .
```

```
/*************************************************************************
 *  After creating the mailbox, the calling task catalogs the mailbox   *
 *  token in the object directory of its own job.                       *
 *************************************************************************/
    CALL RQ$CATALOG$OBJECT        (job,
                                   mbx$token,
                                   @(3, 'MBX'),
                                   @status);

         •
         •  }   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	At least one of the following is true: • The name being cataloged is already in the designated object directory. • The directory's maximum allowable size is 0.
E$EXIST	Either the job parameter (which is not zero) or the object parameter is not a token for an existing object.
E$LIMIT	The designated object directory is full.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The first BYTE of the STRING pointed to by the name parameter contains a value greater than 12 or a value of 0.
E$TYPE	The job parameter is a token for an object which is not a job.

CREATE$COMPOSITE

The CREATE$COMPOSITE system call creates a composite object.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal).

```
composite=RQ$CREATE$COMPOSITE(extension,token$list, except$ptr);
```

INPUT PARAMETERS

extension	A TOKEN for an extension type representing license to create a composite object.
token$list	A POINTER to a structure of the form:

```
Declare
    token$list   STRUCTURE(
    num$slots     WORD,
    num$used      WORD,
    tokens(*)     TOKEN);
```

where:

num$slots	Number of positions available for tokens in token$list.
num$used	Number of component tokens making up the composite object.
token(*)	Tokens that will actually constitute the composite object.

OUTPUT PARAMETERS

composite	A TOKEN to which the Operating System returns the new composite token.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$COMPOSITE system call creates a composite object of the specified extension type. It accepts a list of tokens that specify the component objects and returns a token for the new composite object. A zero value in the token list is a place holder and does not represent an object.

If num$used is less than num$slot, the extra component slots at the end of the composite object are filled with zeros.

If num$slots is less than num$used, the entry list is truncated to fit within the specified number of slots in the composite object.

EXAMPLE

See section "The CREATE_RING_BUFFER Procedure" in Chapter 11.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The extension parameter or one or more of the non-zero token$list parameters does not refer to an existing object.
E$LIMIT	The calling task's job has already reached its object limit.
E$MEM	Insufficient memory is available to satisfy the request.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$PARAM	The specified number of components is zero.
E$TYPE	The extension parameter does not contain a token for an extension object.

CREATE$EXTENSION

The CREATE$EXTENSION system call creates a new object type.

CAUTION

Jobs that create extension objects cannot be deleted until the extension object is deleted. Therefore, you should avoid creating extension objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal).

```
extension=RQ$CREATE$EXTENSION(type$code, deletion$mailbox,
                              except$ptr);
```

INPUT PARAMETERS

type$code	A WORD containing the type code for the new type. The type code for the new type can be any value from 8000H to 0FFFFH and must not be currently in use. (The type codes 0 through 7FFFH are reserved for Intel products.)
deletion$mailbox	A TOKEN for the mailbox where objects of the new type are sent whenever the extension type or their containing job is deleted. A zero value indicates no deletion mailbox is desired.

OUTPUT PARAMETERS

extension	A TOKEN to which the Operating System will return a token for the new type.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$EXTENSION system call returns a token for the newly created extension object type.

You can specify a deletion mailbox when the extension type is created. If you do, a task in your type manager for the new type must wait at the deletion mailbox for tokens of objects of the new extension type that are to be deleted. Tokens of objects are sent to the deletion mailbox for deletion either when their extension type is deleted or when their containing job is deleted; they are not sent there when being deleted by DELETE$COMPOSITE. The task servicing the deletion mailbox may do anything with the composite objects sent to it, but it must delete them.

If you do not want to specify a deletion mailbox, set the token value for deletion$mailbox to zero. If the extension type has no deletion mailbox, composite objects of that type are deleted automatically, and the type manager is not informed. The advantage of having a deletion mailbox is that the type manager has the opportunity to do more than merely delete the composite objects.

A job containing a task that creates an extension object cannot be deleted until the extension object is deleted.

EXAMPLE

See the example in section "The Initialization Part" of Chapter 11.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The calling task's job is partially deleted.
E$EXIST	The deletion$mailbox TOKEN does not refer to an existing object.
E$LIMIT	The calling task's job has reached its object limit.
E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block to satisfy the request.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$PARAM	The type$code parameter is invalid.
E$TYPE	The deletion$mailbox TOKEN does not contain a token for a mailbox.

CREATE$JOB

CREATE$JOB creates a job with a single task.

```
job = RQ$CREATE$JOB (directory$size, param$obj, pool$min, pool$max,
      max$objects, max$tasks, max$priority, except$handler,
      job$flags, task$priority, start$address, data$seg, stack$ptr,
      stack$size, task$flags, except$ptr);
```

INPUT PARAMETERS

directory$size — A WORD specifying the maximum allowable number of entries a job can have in its object directory. The value zero is permitted, for the case where no object directory is desired. The maximum value for this parameter is 0FF0H.

param$obj — A TOKEN indicating the presence or absence of a parameter object. See Chapter 2 for an an explanation of parameter objects.

- if zero, indicates that the new job has no parameter object.
- if not zero, contains a valid token for the new job's parameter object.

pool$min — A WORD which contains the minimum allowable size of the new job's pool, in 16-byte paragraphs. The pool$min parameter is also the initial size of the new job's pool. Pool$min should be at least 32 (decimal). If the stack$ptr parameter has a base value of 0, pool$min should be at least 32 (decimal) plus the value of stack$size in 16 byte paragraphs.

pool$max — A WORD which contains the maximum allowable size of the new job's memory in 16-byte paragraphs. If pool$max is smaller than pool$min, an E$PARAM error occurs.

max$objects — A WORD that specifies the maximum number of objects that the created job can own.

- if not 0FFFFH, contains the maximum number of objects, created by tasks in the new job, that can exist at one time.
- if 0FFFFH, indicates that there is no limit to the number of objects that tasks in the new job can create.

max$tasks	A WORD that specifies the maximum number of tasks that can exist simultaneously in the new job. • If not 0FFFFH, it contains the maximum number of tasks that can exist simultaneously in the new job. • If 0FFFFH, it indicates that there is no limit to the number of tasks that tasks in the new job can create.
max$priority	A BYTE that sets an upper limit on the priority of the tasks created in the new job. • If not zero, it contains the maximum allowable priority of tasks in the new job. If max$priority exceeds the maximum priority of the parent job, an E$LIMIT error occurs. • If zero, it indicates that the new job is to inherit the maximum priority attribute of its parent job.
except$handler	A POINTER to a structure of the following form:

```
STRUCTURE(
    EXCEPTION$HANDLER$PTR      POINTER,
    EXCEPTION$MODE             BYTE);
```

If exception$handler$ptr is not zero, then it is a POINTER to the first instruction of the new job's own exception handler. If exception$handler$ptr is zero, the new job's exception handler is the system default exception handler. In both cases, the exception handler for the new task becomes the default exception handler for the job. The exception$mode indicates when control is to be passed to the exception handler. It is encoded as follows:

Value	When Control Passes To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

job$flags	A WORD containing information that the Nucleus needs to create and maintain the job. The bits (where bit 15 is the high-order bit) have the following meanings:

bit	meaning
15-2	reserved.
1	If 0, then whenever a task in the new job or any of its descendent jobs makes a Nucleus system call, the Nucleus will check the parameters for validity. If 1, the Nucleus will not check the parameters of Nucleus system calls made by tasks in the new job. However, if any ancestor of the new job has been created with this bit set to 0, there will be parameter checking for the new job.
0	reserved.

task$priority — A BYTE that controls the priority of the new job's initial task.

- If not zero, it contains the priority of the new job's initial task. If the task$priority parameter is greater (numerically smaller) than the new job's maximum priority attribute, an E$PARAM error occurs.
- If zero, it indicates that the new job's initial task is to have a priority equal to the new job's maximum priority attribute.

start$address — A POINTER to the first instruction of the new job's initial task (the task created with the job).

data$seg — A WORD or SELECTOR that specifies which data segment the new job is to use.

- If not zero, it contains the base address of the data segment of the new job's initial task.
- If zero, it indicates that the new job's initial task assigns its own data segment. Refer to the iRMX 86 CONFIGURATION GUIDE for more information about data segment allocation.

stack$ptr	A POINTER that specifies the location of the stack for the new job's initial task. • If the base portion is not zero, the pointer points to the base of the user-provided stack of the new job's initial task. • If the base portion is zero, it indicates that the Nucleus should allocate a stack for the new job's initial task. The length of the allocated segment is equal to the value of the stack$size parameter.
stack$size	A WORD containing the size, in bytes, of the stack of the new job's initial task. This size must be at least 16 (decimal) bytes. The Nucleus increases specified values that are not multiples of 16 up to the next higher multiple of 16. The stack size should be at least 300 (decimal) bytes if the new task is going to make Nucleus system calls. Refer to the iRMX 86 PROGRAMMING TECHNIQUES manual for further information on estimating stack sizes.
task$flags	A WORD containing information that the Nucleus needs to create and maintain the job's initial task. The bits (where bit 15 is the high order bit) have the following meanings:

bit	meaning
15-1	Reserved bits which should be set to zero.
0	If one, the initial task contains floating-point instructions. These instructions require the 8087 component for execution. If zero, the initial task does not contain floating-point instructions.

OUTPUT PARAMETERS

job	A TOKEN to which the Operating System will return a token for the new job.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$JOB system call creates a job with an initial task and returns a token for the job. The new job's parent is the calling task's job. The new job counts as one against the parent job's object limit. The new task counts as one against the new job's object and task limits. The new job's resources come from the parent job, as described in the chapter on job management. In particular, the max$task and max$objects values are deducted from the creating job's maximum task and maximum objects attributes, respectively.

EXAMPLE

```
/***********************************************************************
 *  This example illustrates how the CREATE$JOB system call can be    *
 *  used.                                                              *
 ***********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    INITIAL_TASK: PROCEDURE EXTERNAL;
    END INITIAL_TASK;

    DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                   /* if your PL/M compiler does not
                                      support this variable type,
                                      declare TOKEN a WORD */
    DECLARE job$token              TOKEN;
    DECLARE directory$size         WORD;
    DECLARE param$obj              TOKEN;
    DECLARE pool$min               WORD;
    DECLARE pool$max               WORD;
    DECLARE max$objects            WORD;
    DECLARE max$tasks              WORD;
    DECLARE max$priority           BYTE;
    DECLARE except$handler         POINTER;
    DECLARE job$flags              WORD;
    DECLARE task$priority          BYTE;
    DECLARE start$address          POINTER;
    DECLARE data$seg               WORD;
    DECLARE stack$pointer          POINTER;
    DECLARE stack$size             WORD;
    DECLARE task$flags             WORD;
    DECLARE status                 WORD;
```

```
SAMPLE_PROCEDURE:
    PROCEDURE;
    directory$size = 10;          /* max 10 entries in object directory */
    param$obj = 0;                /* new job has no parameter object */
    pool$min = 1FFH;              /* min 1FFH/ max 0FFFFH 16-byte */
    pool$max = 0FFFFH;            /*   paragraphs in job pool */
    max$objects = 0FFFFH;         /* no limit to number of objects */
    max$tasks = 0AH;              /* 0AH tasks can exist simultaneously */
    max$priority = 0;             /* inherit max priority of parent */
    except$handler = 0;           /* use system default except handler */
    job$flags = 0;                /* no flags set */
    task$priority = 0;            /* set initial task to max priority */

    start$address = @INITIAL_TASK;
                                  /* points to first instruction of
                                     initial task */
    data$seg = 0;                 /* initial task sets up own data
                                     segment */
    stack$pointer = 0;            /* Nucleus allocates stack */
    stack$size = 512;             /* 512 bytes in stack of initial task */
    task$flags = 0;               /* no floating-point instructions */
        •
        •  }  Typical PL/M-86 Statements
        •

/**********************************************************************
 *  The calling task creates a job with an initial task labeled        *
 *  INITIAL_TASK.                                                      *
 **********************************************************************/
    job$token = RQ$CREATE$JOB     (directory$size,
                                  param$obj,
                                  pool$min,
                                  pool$max,
                                  max$objects,
                                  max$tasks,
                                  max$priority,
                                  except$handler,
                                  job$flags,
                                  task$priority,
                                  start$address,
                                  data$seg,
                                  stack$pointer,
                                  stack$size,
                                  task$flags,
                                  @status);
        •
        •  }  Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The job containing the calling task is in the process of being deleted.
E$EXIST	The param$obj parameter is not zero and is not a token for an existing object.
E$LIMIT	At least one of the following is true: • max$objects is larger than the unused portion of the object allotment in the calling task's job. • max$tasks is larger than the unused portion of the task allotment in the calling task's job. • max$priority is greater (numerically smaller) than the maximum allowable task priority in the calling task's job. • directory$size is larger than OFFOH. • The new task would exceed the object limit in the new job (that is, the max$objects parameter is set to zero). • The new task would exceed the task limit in the new job (that is, the max$tasks parameter is set to zero).
E$MEM	At least one of the following is true: • The memory available to the new job is not sufficient to create the job descriptor and the object directory. • The memory available to the new job is not sufficient to satisfy the pool$min parameter. • The memory available to the new job is not sufficient to create the task as specified.
E$PARAM	At least one of the following is true: • pool$min is less than 16 + (number of paragraphs needed for the initial task and any system allocated stack) + 5 (if the task uses the 8087 component). • pool$min is greater than pool$max. • task$priority is unequal to zero and greater (numerically smaller) than max$priority.

- stack$size is less than 16.
- pool$max is specified as zero.
- the exception handler mode is not valid.

CREATE$MAILBOX

CREATE$MAILBOX creates a mailbox.

```
mailbox = RQ$CREATE$MAILBOX (mailbox$flags, except$ptr);
```

INPUT PARAMETERS

mailbox$flags — A WORD containing information about the new mailbox. The bits (where bit 15 is the high-order bit) have the following meanings:

bit	meaning
15-5	Reserved bits which should be set to zero.
4-1	A value that, when multiplied by four, specifies the number of objects that can be queued on the high performance object queue. Additional objects are queued on the slower, overflow queue. Four is the minimum size for the high performance queue; that is, specifying zero or one in these bits results in a high performance queue that holds four objects.
0	A bit that determines the queuing scheme for the task queue of the new mailbox, as follows:

value	queueing scheme
0	First-in/first-out
1	Priority based

OUTPUT PARAMETERS

mailbox	A TOKEN to which the Operating System will return a token for the new mailbox.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$MAILBOX system call creates a mailbox and returns a token for it. The new mailbox counts as one against the object limit of the calling task's job.

EXAMPLE

```
/**********************************************************************
 *  This example illustrates how the CREATE$MAILBOX system call can be  *
 *  used.                                                               *
 **********************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

   DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                  /* if your PL/M compiler does not
                                     support this variable type,
                                     declare TOKEN a WORD */
   DECLARE mbx$token              TOKEN;
   DECLARE mbx$flags              WORD;
   DECLARE status                 WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

   mbx$flags = 0;                 /* designates four objects to be queued
                                     on the high performance object
                                     queue; designates a first-in/
                                     first-out task queue. */

         •
         •    Typical PL/M-86 Statements
         •

/**********************************************************************
 *  The token mbx$token is returned when the calling task invokes the  *
 *  CREATE$MAILBOX system call.                                         *
 **********************************************************************/
   mbx$token = RQ$CREATE$MAILBOX (mbx$flags,
                                  @status);

         •
         •    Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditons.
E$LIMIT	The requested mailbox would exceed the job object limit of the calling task's job.
E$MEM	The memory available to the calling task's job is not sufficient to create a mailbox.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

CREATE$REGION

The CREATE$REGION system call creates a region.

CAUTION

Tasks which use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal) while the task is in the region.

```
region = RQ$CREATE$REGION (region$flags, except$ptr);
```

INPUT PARAMETER

region$flags	A WORD that specifies the queueing protocol of the new region. If the low order bit equals zero, tasks await access in FIFO order. If the low order bit equals one, tasks await access in priority order. The other bits in the WORD are reserved and should be set to zero.

OUTPUT PARAMETERS

region	A TOKEN to which the Operating System will return a token for the new region.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$REGION system call creates a region and returns a token for the region.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the CREATE$REGION system call can be    *
 *  used.                                                                *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE region$token               TOKEN;
    DECLARE priority$queue             LITERALLY '1'; /* tasks wait in
                                                         priority order */
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

         •
         •    Typical PL/M-86 Statements
         •

/**************************************************************************
 *  The token region$token is returned when the calling task invokes     *
 *  the CREATE$REGION system call.                                       *
 **************************************************************************/
    region$token = RQ$CREATE$REGION  (priority$queue,
                                       @status);

         •
         •    Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	The calling task's job has reached its object limit.
E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block to satisfy the request.
E$NOT$CONFIGURED	This system call is not part of the present configuration.

CREATE$SEGMENT

CREATE$SEGMENT creates a segment.

```
segment = RQ$CREATE$SEGMENT (size, except$ptr);
```

INPUT PARAMETER

size	A WORD that specifies the size of the requested segment. • If not zero, it contains the size, in bytes, of the requested segment. If the size parameter is not a multiple of 16, it will be rounded up to the nearest higher multiple of 16 before the request is processed by the Nucleus. • If zero, it indicates that the size of the request is 65536 (64K) bytes.

OUTPUT PARAMETERS

segment	A TOKEN to which the Operating System will return a token for the new segment.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$SEGMENT system call creates a segment and returns the token for it. The memory for the segment is taken from the free portion of the memory pool of the calling task's job, unless borrowing from the parent job is both necessary and possible. The new segment counts as one against the object limit of the calling task's job.

To gain access into the segment, you should base an array or structure on a pointer by setting the base portion equal to the segment's TOKEN and the offset portion equal to zero. If you have a PL/M-86 compiler that supports the SELECTOR data type, you can accomplish the same thing by basing the array or structure on the SELECTOR.

EXAMPLE

```
/***************************************************************************
 *  This example illustrates how the CREATE$SEGMENT system call can be   *
 *  used.                                                                 *
 ***************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                   /* if your PL/M compiler does not
                                      support this variable type,
                                      declare TOKEN a WORD */
    DECLARE seg$token              TOKEN;
    DECLARE seg$size               WORD;
    DECLARE status                 WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    seg$size = 0100H;              /* the size of the requested segment is
                                      256 bytes. */

        •
        •  }   Typical PL/M-86 Statements
        •

/***************************************************************************
 *  The token seg$token is returned when the calling task invokes the    *
 *  CREATE$SEGMENT system call.                                           *
 ***************************************************************************/
    seg$token = RQ$CREATE$SEGMENT (seg$size,
                                   @status);

        •
        •  }   Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	The requested segment would exceed the job object limit.
E$MEM	The memory available to the calling task's job is not sufficient to create the specified segment.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

SYSTEM CALLS

CREATE$SEMAPHORE

CREATE$SEMAPHORE creates a semaphore.

```
semaphore = RQCREATE$SEMAPHORE (initial$value, max$value,
                                semaphore$flags, except$ptr);
```

INPUT PARAMETERS

initial$value	A WORD containing the initial number of units to be in the custody of the new semaphore.
max$value	A WORD containing the maximum number of units over which the new semaphore is to have custody at any given time. If max$value is zero, an E$PARAM error occurs.
semaphore$flags	A WORD containing information about the new semaphore. The low-order bit determines the queueing scheme for the new semaphore's task queue:

Value	Queueing Scheme
0	First-in/first-out
1	Priority based

The remaining bits in semaphore$flags are reserved for future use and should be set to zero.

OUTPUT PARAMETERS

semaphore	A TOKEN to which the Operating System will return a token for the new semaphore.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$SEMAPHORE system call creates a semaphore and returns a token for it. The semaphore thus created counts as one against the object limit of the calling task's job.

EXAMPLE

```
/**********************************************************************
 *  This example illustrates how the CREATE$SEMAPHORE system call can  *
 *  be used.                                                           *
 **********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE TOKEN                   LITERALLY 'SELECTOR';
                                    /* if your PL/M compiler does not
                                       support this variable type,
                                       declare TOKEN a WORD */
    DECLARE sem$token               TOKEN;
    DECLARE init$value              WORD;
    DECLARE max$value               WORD;
    DECLARE sem$flags               WORD;
    DECLARE status                  WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    init$value = 1;                 /* the new semaphore has one initial
                                       unit */
    max$value = 10H;                /* the new semaphore can have a maximum
                                       of 16 units */
    sem$flags = 0;                  /* designates a first-in/
                                       first-out task queue. */
            .
            .  }    Typical PL/M-86 Statements
            .

/**********************************************************************
 *  The token sem$token is returned when the calling task invokes the  *
 *  CREATE$SEMAPHORE system call.                                      *
 **********************************************************************/
    sem$token = RQ$CREATE$SEMAPHORE  (init$value,
                                      max$value,
                                      sem$flags,
                                      @status);
            .
            .  }    Typical PL/M-86 Statements
            .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	The requested semaphore would exceed the job object limit.
E$MEM	The memory available to the calling task's job is not sufficient to create a semaphore.
E$PARAM	At least one of the following is true: • The initial$value parameter is larger than the maximum$value parameter. • The maximum$value parameter is 0.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

CREATE$TASK

CREATE$TASK creates a task.

```
task = RQ$CREATE$TASK (priority, start$address, data$seg, stack$ptr,
                       stack$size, task$flags, except$ptr);
```

INPUT PARAMETERS

priority	A BYTE that specifies the priority of the new task. • If not zero, it contains the priority of the new task. The priority parameter must not exceed the maximum allowable priority of the calling task's job. If it does, an E$PARAM error occurs. • If zero, it indicates that the new task's priority is to equal the maximum allowable priority of the calling task's job.
start$address	A POINTER to the first instruction of the new task.
data$seg	A WORD or SELECTOR that specifies the new task's data segment. • If not zero, the WORD contains the base address of the new task's data segment. • If zero, the WORD indicates that the new task assigns its own data segment. Refer to the iRMX 86 CONFIGURATION GUIDE for further information on data segment allocation.
stack$ptr	A POINTER that specifies the location of the stack for the new task. • If the base portion is not zero, the POINTER points to the base of the new task's stack. • If the base portion is zero, the Nucleus allocates a stack to the new task. The length of the stack is equal to the value of the stack$size parameter.
stack$size	A WORD containing the size, in bytes, of the new task's stack segment. The stack size must be at least 16 bytes. The Nucleus increases specified values that are not multiples of 16 up to the next higher multiple of 16.

The stack size should be at least 300 bytes if the new task is going to make Nucleus system calls. Refer to the iRMX 86 PROGRAMMING TECHNIQUES manual for further information on assigning stack sizes.

task$flags	A WORD containing information that the Nucleus needs to create and maintain the task. The bits (where bit 15 is the high-order bit) have the following meanings:

bit	meaning
15-1	Reserved bits which should be set to zero.
0	If one, the task contains floating-point instructions. These instructions require the 8087 component for execution. If zero, the task does not contain floating-point instructions.

OUTPUT PARAMETERS

task	A TOKEN to which the Operating System will return a token for the new task.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The CREATE$TASK system call creates a task and returns a token for it. The new task counts as one against the object and task limits of the calling task's job. Attributes of the new task are initialized upon creation as follows:

- priority: as specified in the call.
- execution state: ready.
- suspension depth: 0.
- containing job: the job which contains the calling task.
- exception handler: the exception handler of the containing job.
- exception mode: the exception mode of the containing job.

EXAMPLE

```
/*************************************************************************
 *  This example illustrates how the CREATE$TASK system call can be     *
 *  used.                                                                *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    TASK_CODE: PROCEDURE EXTERNAL;
    END TASK_CODE;

    DECLARE TOKEN                   LITERALLY 'SELECTOR';
                                    /* if your PL/M compiler does not
                                       support this variable type,
                                       declare TOKEN a WORD */
    DECLARE task$token              TOKEN;
    DECLARE priority$level$66       LITERALLY '66';
    DECLARE start$address           POINTER;
    DECLARE data$seg                WORD;
    DECLARE stack$pointer           POINTER;
    DECLARE stack$size$512          LITERALLY '512'; /* new task's stack
                                                        size is 512 bytes */
    DECLARE task$flags              WORD;
    DECLARE status                  WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;
    start$address = @TASK_CODE;     /* first instruction of the new task */
    data$seg = 0;                   /* task sets up own data segment */
    stack$pointer = 0;              /* automatic stack allocation  */
    task$flags = 0;                 /* designates no floating-point
                                       instructions */
          .
          .  }  Typical PL/M-86 Statements
          .
```

```
/************************************************************************
 *  The task (whose code is labeled TASK_CODE) is created when the     *
 *  calling task invokes the CREATE$TASK system call.                  *
 ************************************************************************/
   task$token = RQ$CREATE$TASK    (priority$level$66,
                                   start$address,
                                   data$seg,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);
        •
        •  }    Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	At least one of the following is true: • The new task would exceed the object limit or the task limit of the calling task's job. • The priority parameter is nonzero and greater (numerically smaller) than the maximum allowable priority for tasks in the calling task's job.
E$MEM	The memory available to the calling task's job is not sufficient to create a task as specified (task descriptor, stack, and possibly 8087 area).
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The stack$size parameter is less than 16.

DELETE$COMPOSITE

The DELETE$COMPOSITE system call deletes a composite object.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal).

```
CALL RQ$DELETE$COMPOSITE(extension, composite, except$ptr);
```

INPUT PARAMETERS

extension	A TOKEN for the extension type used as a license to create the composite object to be deleted.
composite	A TOKEN for the composite object to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$COMPOSITE system call deletes the specified composite object but not its component objects.

EXAMPLE

See the example in section "The Initialization Part" of Chapter 11.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The extension type does not match the composite parameter.
E$EXIST	One or both of the extension or composite parameters does not refer to a currently existing object.
E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block for Nucleus housekeeping purposes.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	One or both of the extension or composite parameters is not of the correct object type.

DELETE$EXTENSION

The DELETE$EXTENSION system call deletes an extension object and all composites of that type.

CAUTION

Jobs that create extension objects cannot be deleted until the extension object is deleted. Therefore, you should avoid creating extension objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal).

```
CALL RQ$DELETE$EXTENSION(extension, except$ptr);
```

INPUT PARAMETER

extension	A TOKEN for the extension object to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$EXTENSION system call deletes the specified extension object type and all composite objects of that type. This makes the corresponding type code available for reuse.

If a deletion mailbox was specified when the extension type was created, then all of the composite objects created by the extension type to be deleted are sent to that deletion mailbox. In this case, this call will not be completed until all of the composite objects have been deleted.

If the extension type has no deletion mailbox, the composite objects created by the extension type to be deleted are deleted without informing the type manager.

The job containing the task that created the extension object type cannot be deleted until the extension object is deleted.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The extension parameter does not refer to an existing object.
E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block for Nucleus housekeeping purposes.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	The extension parameter does not contain a token for an extension object.

DELETE$JOB

DELETE$JOB deletes a job.

```
CALL RQ$DELETE$JOB (job, except$ptr);
```

INPUT PARAMETER

job	A TOKEN for the job to be deleted. A value of zero specifies the calling task's job.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$JOB system call deletes from the system the specified job, as well as all objects created by tasks in it. Exceptions are that jobs and extension objects (see Chapter 11) created by tasks in the target job must be deleted prior to the call to DELETE$JOB. Information concerning the descendents of a job is obtained via the OFFSPRING system call. During deletion, all resources that the target job had borrowed from its parent are returned.

Deleting a job causes a credit of one toward the object total of the parent job. Also, the maximum tasks and maximum objects attributes of the deleted job are credited to the current tasks and current objects attributes, respectively, of the parent job.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the DELETE$JOB system call can be      *
 *  used to delete the calling task's job.                              *
 ************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

   DECLARE calling$tasks$job          LITERALLY '0';
   DECLARE status                     WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

         •
         •   Typical PL/M-86 Statements
         •

/************************************************************************
 *  If you set the selection parameter to zero, the DELETE$JOB system   *
 *  call will delete the calling task's job.                            *
 ************************************************************************/
   CALL RQ$DELETE$JOB                 (calling$tasks$job,
                                      @status);

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	At least one of the following is true: • There are undeleted jobs, or extension objects (see Chapter 11) which have been created by tasks in the target job. • The deleting task has access to data guarded by a region contained in the job to be deleted. (Refer to Chapter 9 for information concerning regions.)
E$EXIST	The job parameter is not a token for an existing object.
E$MEM	The job to be deleted contains undeleted composite objects (see Chapter 11), and there is not sufficient memory for the Nucleus to send deletion messages to the appropriate deletion mailboxes.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The job parameter is a token for an object that is not a job.

DELETE$MAILBOX

DELETE$MAILBOX deletes a mailbox.

```
CALL RQ$DELETE$MAILBOX (mailbox, except$ptr);
```

INPUT PARAMETER

mailbox	A TOKEN for the mailbox to be deleted.

OUTPUT PARAMETERS

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$MAILBOX system call deletes the specified mailbox. If any tasks are queued at the mailbox at the moment of deletion, they are awakened with an E$EXIST exceptional condition. If there is a queue of object tokens at the moment of deletion, the queue is discarded. Deleting the mailbox counts as a credit of one toward the object total of the containing job.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the DELETE$MAILBOX system call can be  *
 *  used.                                                               *
 ************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE TOKEN                   LITERALLY 'SELECTOR';
                                    /* if your PL/M compiler does not
                                       support this variable type,
                                       declare TOKEN a WORD */
    DECLARE mbx$token               TOKEN;
    DECLARE mbx$flags               WORD;
    DECLARE status                  WORD;
```

```
SAMPLE_PROCEDURE:
    PROCEDURE;

    mbx$flags = 0;                  /* designates four objects to be queued
                                       on the high performance object
                                       queue;  designates a first-in/
                                       first-out task queue. */

        •
        •  }  Typical PL/M-86 Statements
        •

/************************************************************************
 *  In order to delete a mailbox, a task must know the token for that   *
 *  mailbox.  In this example, the needed token is known because the    *
 *  calling task creates the mailbox.                                   *
 ************************************************************************/
    mbx$token = RQ$CREATE$MAILBOX (mbx$flags,
                                   @status);

        •
        •  }  Typical PL/M-86 Statements
        •

/************************************************************************
 *  When the mailbox is no longer needed, it may be deleted by any task *
 *  that knows the token for the mailbox.                               *
 ************************************************************************/

    CALL RQ$DELETE$MAILBOX         (mbx$token,
                                   @status);

        •
        •  }  Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	Either the mailbox parameter is not a token for an existing object or it represents a mailbox whose job is in the process of being deleted.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The mailbox parameter is a token for an object which is not a mailbox.

DELETE$REGION

The DELETE$REGION system call deletes a region.

CAUTION

Tasks which use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal) while the task is in the region.

```
CALL RQ$DELETE$REGION (region, except$ptr);
```

INPUT PARAMETER

region	A TOKEN for the region to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$REGION system call deletes a region. If a task that has access to data protected by the region requests that that region be deleted, the task receives an E$CONTEXT exceptional condition. If a task requests deletion while another task has access, deletion is delayed until access is surrendered. When the region is deleted, any waiting tasks awaken with an E$EXIST exceptional condition.

EXAMPLE

```
/*************************************************************************
 *  This example illustrates how the DELETE$REGION system call can be   *
 *  used.                                                                *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE region$token               TOKEN;
    DECLARE priority$queue             LITERALLY '1'; /* tasks wait in
                                                         priority order */
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;
            •
            •  }   Typical PL/M-86 Statements
            •

/*************************************************************************
 *  In order to delete a region, a task must know the token for that    *
 *  region.  In this example, the needed token is known because the     *
 *  calling task creates the region.                                     *
 *************************************************************************/

    region$token = RQ$CREATE$REGION   (priority$queue,
                                       @status);
            •
            •  }   Typical PL/M-86 Statements
            •

/*************************************************************************
 *  When the region is no longer needed, it may be deleted by any task  *
 *  that knows the token for the region.                                 *
 *************************************************************************/

    CALL RQ$DELETE$REGION              (region$token,
                                       @status);
            •
            •  }   Typical PL/M-86 Statements
            •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The deletion is being requested by a task that currently holds access to data protected by the region.
E$EXIST	The region does not refer to an existing object.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	The region parameter is a token for an object that is not a region.

DELETE$SEGMENT

DELETE$SEGMENT deletes a segment.

```
CALL RQ$DELETE$SEGMENT (segment, except$ptr);
```

INPUT PARAMETER

segment	A TOKEN for the segment that is to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$SEGMENT system call returns the specified segment to the memory pool from which it was allocated. The deleted segment counts as a credit of one toward the object total of the containing job.

EXAMPLE

```
/*************************************************************************
 *  This example illustrates how the DELETE$SEGMENT system call can be   *
 *  used.                                                                 *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);        /* Declares all system calls */

    DECLARE TOKEN                    LITERALLY 'SELECTOR';
                                     /* if your PL/M compiler does not
                                        support this variable type,
                                        declare TOKEN a WORD */
    DECLARE seg$token                TOKEN;
    DECLARE size                     WORD;
    DECLARE status                   WORD;
```

```
SAMPLE_PROCEDURE:
    PROCEDURE;
    size = 64;                      /* designates new segment to contain
                                       64 bytes */

        •
        •  }    Typical PL/M-86 Statements
        •

/*************************************************************************
 *  In order to delete a segment, a task must know the token for that   *
 *  segment.  In this example, the needed token is known because the    *
 *  calling task creates the segment.                                   *
 *************************************************************************/

    seg$token = RQ$CREATE$SEGMENT   (size,
                                    @status);
        •
        •  }    Typical PL/M-86 Statements
        •

/*************************************************************************
 *  When the segment is no longer needed, it may be deleted by any task *
 *  that knows the token for the segment.                               *
 *************************************************************************/

    CALL RQ$DELETE$SEGMENT          (seg$token,
                                    @status);
        •
        •  }    Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	Either the segment parameter is not a token for an existing object or it represents a segment whose job is in the process of being deleted.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The segment parameter is a token for an object that is not a segment.

DELETE$SEMAPHORE

DELETE$SEMAPHORE deletes a semaphore.

```
CALL RQ$DELETE$SEMAPHORE (semaphore, except$ptr);
```

INPUT PARAMETER

semaphore	A TOKEN for the semaphore that is to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$SEMAPHORE system call deletes the specified semaphore. If there are tasks in the semaphore's queue at the moment of deletion, they are awakened with an E$EXIST exceptional condition. The deleted semaphore counts as a credit of one toward the object total of the containing job.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the DELETE$SEMAPHORE system call can    *
 *  be used.                                                             *
 ************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE TOKEN                   LITERALLY 'SELECTOR';
                                    /* if your PL/M compiler does not
                                       support this variable type,
                                       declare TOKEN a WORD */
    DECLARE sem$token               TOKEN;
    DECLARE init$value              WORD;
    DECLARE max$value               WORD;

    DECLARE sem$flags               WORD;
    DECLARE status                  WORD;
```

```
SAMPLE_PROCEDURE:
    PROCEDURE;

    init$value = 1;              /* the new semaphore has one initial
                                    unit */
    max$value = 10H;             /* the new semaphore can have a maximum
                                    of 16 units */
    sem$flags = 0;               /* designates a first-in/
                                    first-out task queue. */
         •
         •   }   Typical PL/M-86 Statements
         •

/***********************************************************************
 *  In order to delete a semaphore, a task must know the token for that *
 *  semaphore.  In this example, the needed token is known because the  *
 *  calling task creates the semaphore.                                 *
 ***********************************************************************/

    sem$token = RQ$CREATE$SEMAPHORE  (init$value,
                                     max$value,
                                     sem$flags,
                                     @status);
         •
         •   }   Typical PL/M-86 Statements
         •

/***********************************************************************
 *  When the semaphore is no longer needed, it may be deleted by any    *
 *  task that know the token for the semaphore.                         *
 ***********************************************************************/

    CALL RQ$DELETE$SEMAPHORE         (sem$token,
                                     @status);
         •
         •   }   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	Either the semaphore parameter is not a token for an existing object or it represents a semaphore whose job is in the process of being deleted.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The semaphore parameter is a token for an object that is not a semaphore.

DELETE$TASK

DELETE$TASK deletes a task.

```
CALL RQ$DELETE$TASK (task, except$ptr);
```

INPUT PARAMETER

task	A TOKEN that identifies the task to be deleted. • If not zero, the TOKEN contains a token for the task that is to be deleted. • If zero, the iRMX 86 Operating System will delete the calling task.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DELETE$TASK system call deletes the specified task from the system and from any queues in which the task was waiting. Deleting the task counts as a credit of one toward the object total of the containing job. It also counts as a credit of one toward the containing job's task total.

You cannot successfully delete an interrupt task by invoking this system call. Any attempt to do so results in an E$CONTEXT exceptional condition. To delete an interrupt task, invoke the RESET$INTERRUPT system call.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the DELETE$TASK system call can be      *
 *  used.                                                                 *
 **************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

   TASK_CODE: PROCEDURE EXTERNAL;
   END TASK_CODE;

   DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                  /* if your PL/M compiler does not
                                     support this variable type,
                                     declare TOKEN a WORD */
   DECLARE task$token             TOKEN;
   DECLARE priority$level$66      LITERALLY '66';
   DECLARE start$address          POINTER;
   DECLARE data$seg               WORD;
   DECLARE stack$pointer          POINTER;
   DECLARE stack$size$512         LITERALLY '512'; /* new task's stack
                                                      size is 512 bytes */
   DECLARE task$flags             WORD;
   DECLARE status                 WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

   start$address = @TASK_CODE;    /* points to first instruction of
                                     the new task */
   data$seg = 0;                  /* task sets up own data segment */
   stack$pointer = 0;             /* automatic stack allocation  */
   task$flags = 0;                /* indicates no floating-point
                                     instructions */
         •
         •  }    Typical PL/M-86 Statements
         •

/**************************************************************************
 *  In order to delete a task, a task must know the token for that       *
 *  task.  In this example, the needed token is known because the        *
 *  calling task creates the new task (The task's code is labeled        *
 *  TASK_CODE).                                                           *
 **************************************************************************/

   task$token = RQ$CREATE$TASK    (priority$level$66,
                                  start$address,
                                  data$seg,
                                  stack$pointer,
                                  stack$size$512,
                                  task$flags,
                                  @status);
```

```
          .
          .  }  Typical PL/M-86 Statements
          .

/*********************************************************************
 *  The calling task has created a task (whose code is labeled       *
 *  TASK_CODE) which is not an interrupt task.  When this task is no *
 *  longer needed, it may be deleted by any task that knows its token. *
 *********************************************************************/

    CALL RQ$DELETE$TASK           (task$token,
                                   @status);
          .
          .  }  Typical PL/M-86 Statements
          .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The task parameter is a token for an interrupt task.
E$EXIST	Either the task parameter is not a token for an existing object or it represents a task whose job is in the process of being deleted.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The task parameter is a token for an object which is not a task.

DISABLE

DISABLE disables an interrupt level.

```
CALL RQ$DISABLE (level, except$ptr);
```

INPUT PARAMETER

level — A WORD that specifies an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call. All exceptional conditions must be processed in-line. Control does not pass to an exception handler.

DESCRIPTION

The DISABLE system call disables the specified interrupt level. It has no effect on other levels. The level must have an interrupt handler assigned to it. The level reserved for the system clock should not be disabled. This level is determined during system configuration (refer to the iRMX 86 CONFIGURATION GUIDE).

EXAMPLE

```
/***********************************************************************
 *  This example illustrates how the DISABLE system call can be used to *
 *  disable an interrupt level.                                          *
 ***********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    INTERRUPT_HANDLER: PROCEDURE EXTERNAL;
    END INTERRUPT_HANDLER;

    DECLARE interrupt$level$7      LITERALLY '0000 0000 0111 1000B';
                                   /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag    BYTE;
    DECLARE interrupt$handler      POINTER;
    DECLARE data$segment           WORD;
    DECLARE status                 WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    interrupt$task$flag = 0;       /* indicates no interrupt task on level
                                      7 */
    data$segment = 0;              /* indicates that interrupt handler
                                      will load its own data segment */
    interrupt$handler = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                   /* points to first instruction of
                                      interrupt handler */
        .
        .   Typical PL/M-86 Statements
        .

/***********************************************************************
 *  An interrupt level must have an interrupt handler or an interrupt   *
 *  task assigned to it.  Invoking the SET$INTERRUPT system call, the   *
 *  calling task assigns INTERRUPT_HANDLER to interrupt level 7.         *
 ***********************************************************************/

    CALL RQ$SET$INTERRUPT         (interrupt$level$7,
                                   interrupt$task$flag,
                                   interrupt$handler,
                                   data$segment,
                                   @status);
        .
        .   Typical PL/M-86 Statements
        .
```

```
/************************************************************************
 *  The SET$INTERRUPT system call enabled interrupt level 7.  In order  *
 *  to disable level 7, the calling task invokes the DISABLE system     *
 *  call.                                                               *
 ************************************************************************/

   CALL RQ$DISABLE                (interrupt$level$7,
                                  @status);
        •
        •}    Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The level indicated by the level parameter is already disabled.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

DISABLE$DELETION

The DISABLE$DELETION system call makes an object immune to ordinary deletion.

CAUTION

DISABLE$DELETION makes an object immune to ordinary deletion by increasing the disabling depth of an object. If a Human Interface application contains objects whose disabling depths are greater than one, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal). Therefore you should not use DISABLE$DELETION (and have no need to use ENABLE$DELETION or FORCE$DELETE) in Human Interface applications.

```
CALL RQ$DISABLE$DELETION (object, except$ptr);
```

INPUT PARAMETER

object	A TOKEN for the object whose deletion is to be disabled.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The DISABLE$DELETION system call increases by one the disabling depth of an object, making it immune to ordinary deletion and possibly making it immune to forced deletion. If a task attempts to delete the object while it is immune, the task sleeps until the immunity is removed. At that time, the object is deleted and the task is awakened.

NOTES

If an object within a job has had its deletion disabled then the containing job cannot be deleted until that object has had its deletion reenabled.

An attempt to raise an object's disabling depth above 255 causes an E$LIMIT exceptional condition.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the DISABLE$DELETION system call can    *
 *  be used to make an object immune to ordinary deletion.               *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);         /* Declares all system calls */

    DECLARE TOKEN                     LITERALLY 'SELECTOR';
                                      /* if your PL/M compiler does not
                                         support this variable type,
                                         declare TOKEN a WORD */
    DECLARE task$token                TOKEN;
    DECLARE calling$task              LITERALLY '0';
    DECLARE status                    WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

            •
            •   }   Typical PL/M-86 Statements
            •

/**************************************************************************
 *  In this example the calling task will be the object to become        *
 *  immune to ordinary deletion.  The GET$TASK$TOKEN is invoked by the   *
 *  calling task to obtain its own token.                                *
 **************************************************************************/

    task$token = RQ$GET$TASK$TOKENS  (calling$task,
                                      @status);
            •
            •   }   Typical PL/M-86 Statements
            •

/**************************************************************************
 *  Using its own token, the calling task invokes the DISABLE$DELETION   *
 *  system call to increase its own disabling depth by one.  This makes  *
 *  the calling task immune to deletion.                                 *
 **************************************************************************/
```

```
          CALL RQ$DISABLE$DELETION     (task$token,
                                        @status);
                •
                •  }   Typical PL/M-86 Statements
                •

      END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The object parameter does not refer to an existing object.
E$LIMIT	The object's disabling depth is already 255.
E$NOT$CONFIGURED	This system call is not part of the present configuration.

ENABLE

ENABLE enables an interrupt level.

```
CALL RQ$ENABLE (level, except$ptr);
```

INPUT PARAMETER

level — A WORD that specifies an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The ENABLE system call enables the specified interrupt level. The level must have an interrupt handler assigned to it. A task must not enable the level associated with the system clock.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the ENABLE system call can be used to  *
 *  enable an interrupt level.                                          *
 ************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    INTERRUPT_HANDLER: PROCEDURE EXTERNAL;
    END INTERRUPT_HANDLER;

    DECLARE interrupt$level$7      LITERALLY '0000 0000 0111 1000B';
                                   /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag    BYTE;
    DECLARE interrupt$handler      POINTER;
    DECLARE data$segment           WORD;
    DECLARE status                 WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    interrupt$task$flag = 0;       /* indicates no interrupt task on level
                                      7 */
    data$segment = 0;              /* indicates that interrupt handler
                                      will load its own data segment */
    interrupt$handler = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                   /* points to first instruction of
                                      interrupt handler */
          .
          .  }    Typical PL/M-86 Statements
          .

/************************************************************************
 *  An interrupt level must have an interrupt handler or an interrupt   *
 *  task assigned to it.  Invoking the SET$INTERRUPT system call, the   *
 *  calling task assigns INTERRUPT_HANDLER to interrupt level 7.        *
 ************************************************************************/

    CALL RQ$SET$INTERRUPT          (interrupt$level$7,
                                   interrupt$task$flag,
                                   interrupt$handler,
                                   data$segment,
                                   @status);
          .
          .  }    Typical PL/M-86 Statements
          .
```

```
/************************************************************************
 *  The SET$INTERRUPT system call enabled interrupt level 7.  In order  *
 *  to illustrate the use of the ENABLE system call, interrupt level 7  *
 *  must first be disabled.  The calling task invokes the DISABLE       *
 *  system call to disable interrupt level 7.                           *
 ************************************************************************/

   CALL RQ$DISABLE                (interrupt$level$7,
                                   @status);
         •
         •   Typical PL/M-86 Statements
         •

/************************************************************************
 *  When an interrupt level needs to be enabled, a task must invoke the *
 *  ENABLE system call.                                                 *
 ************************************************************************/

   CALL RQ$ENABLE                 (interrupt$level$7,
                                   @status);
         •
         •   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	At least one of the following is true: • A non-interrupt task tried to enable a level that was already enabled. • There is not an interrupt handler assigned to the specified level. • There has been an interrupt overflow on the specified level.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

ENABLE$DELETION

The ENABLE$DELETION system call enables the deletion of objects that have had deletion disabled.

CAUTION

DISABLE$DELETION makes an object immune to ordinary deletion by increasing the disabling depth of an object. If a Human Interface application contains objects whose disabling depths are greater than one, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal). Therefore you should not use DISABLE$DELETION (and have no need to use ENABLE$DELETION or FORCE$DELETE) in Human Interface applications.

```
CALL RQ$ENABLE$DELETION (object, except$ptr);
```

INPUT PARAMETER

object	A TOKEN for the object whose deletion is to be enabled.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The ENABLE$DELETION system call decreases by one the disabling depth of an object. If there is a pending deletion request against the object, and the ENABLE$DELETION call makes the object eligible for deletion, the object is deleted and the task which made the deletion request is awakened.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the ENABLE$DELETION system call can be *
 *  used to enable the deletion of a task that had been deletion        *
 *  disabled.                                                            *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE task$token                 TOKEN;
    DECLARE calling$task               LITERALLY '0';
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

         •
         •   } Typical PL/M-86 Statements
         •

/**************************************************************************
 *  In this example the calling task will be the object to become        *
 *  immune to deletion.  The GET$TASK$TOKEN is invoked by the calling    *
 *  task to obtain its own token.                                        *
 *************************************************************************/

    task$token = RQ$GET$TASK$TOKENS    (calling$task,
                                       @status);

         •
         •   } Typical PL/M-86 Statements
         •

/**************************************************************************
 *  Using its own token, the calling task invokes the DISABLE$DELETION   *
 *  system call to increase its own disabling depth by one.  This makes *
 *  the calling task immune to deletion.                                 *
 *************************************************************************/

    CALL RQ$DISABLE$DELETION           (task$token,
                                       @status);

         •
         •   } Typical PL/M-86 Statements
         •

/**************************************************************************
 *  In order to allow itself to be deleted, the calling task invokes     *
 *  the ENABLE$DELETION system call.  This system call decreases by one *
 *  the disabling depth of an object.  In this example, the object is   *
 *  the calling task.                                                    *
 *************************************************************************/
```

```
          CALL RQ$ENABLE$DELETION           (task$token,
                                             @status);
               •
               •    Typical PL/M-86 Statements
               •

      END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The object's deletion is not disabled.
E$EXIST	The object parameter does not refer to an existing object.
E$NOT$CONFIGURED	This system call is not part of the present configuration.

END$INIT$TASK

END$INIT$TASK is used by an initialization task to inform the root task that it has completed its synchronous initialization process.

```
    CALL RQ$END$INIT$TASK;
```

DESCRIPTION

When the initialization task finishes its synchronous initialization, it must inform the root task that it is finished, so that the root task can resume execution and create another first-level job. When you call END$INIT$TASK, the root task resumes execution, allowing it to create the next first-level job. You must include this system call in the initialization task of each first-level job, even if the jobs require no synchronous initialization. Refer to the iRMX 86 CONFIGURATION GUIDE for more information on first-level jobs and the initialization process.

ENTER$INTERRUPT

ENTER$INTERRUPT is used by interrupt handlers to load a previously specified segment base address into the DS register.

```
CALL RQ$ENTER$INTERRUPT(level, except$ptr);
```

INPUT PARAMETER

level — A WORD specifying an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call. All exceptional conditions must be processed in-line. Control does not pass to an exception handler.

DESCRIPTION

ENTER$INTERRUPT, on behalf of the calling interrupt handler, loads a base address value into the DS register. The value is what was specified when the interrupt handler was set up by an earlier call to SET$INTERRUPT.

If the handler is going to call an interrupt task, ENTER$INTERRUPT allows the handler to place data in the iAPX 86 data segment that will be used by the interrupt task. This provides a mechanism for the interrupt handler to pass data to the interrupt task.

EXAMPLE

```
/***********************************************************************
 *  This example illustrates how the ENTER$INTERRUPT system call can be *
 *  used to load a segment base address into the data segment register. *
 ***********************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

   DECLARE the$first$word         WORD;
   DECLARE interrupt$level$7      LITERALLY '0000 0000 0111 1000B';
                                  /* specifies master interrupt level 7 */
   DECLARE interrupt$task$flag    BYTE;
   DECLARE intrpt$handlr$addrs    POINTER;
   DECLARE data$segment           WORD;
   DECLARE status                 WORD;
   DECLARE interrupt$status       WORD;
   DECLARE ds$pointer             POINTER;
   DECLARE PTR$OVERLAY            LITERALLY 'STRUCTURE (offset   WORD,
                                                        base     WORD)';
                                  /* establishes a structure for
                                     overlays */
   DECLARE ds$pointer$ovly        PTR$OVERLAY AT (@ds$pointer);
                                  /* using the overlay structure, the
                                     base address of the interrupt
                                     handler's data segment is
                                     identified */

INTERRUPT_HANDLER: PROCEDURE INTERRUPT 59 PUBLIC;

         •
         •  }    Typical PL/M-86 Statements
         •

/***********************************************************************
 *  The calling interrupt handler invokes the ENTER$INTERRUPT system    *
 *  call which loads a base address value (defined by                   *
 *  ds$pointer$ovly.base) into the data segment register.               *
 ***********************************************************************/

   CALL RQ$ENTER$INTERRUPT        (interrupt$level$7,
                                  @interrupt$status);
   CALL INLINE_ERROR_PROCESS      (interrupt$status);

         •
         •  }    Typical PL/M-86 Statements
         •
```

```
/*************************************************************************
 *  Interrupt handlers that do not invoke interrupt tasks need to        *
 *  invoke the EXIT$INTERRUPT system call to send an end-of-interrupt    *
 *  signal to the hardware.                                              *
 ************************************************************************/
    CALL RQ$EXIT$INTERRUPT          (interrupt$level$7,
                                    @interrupt$status);
    CALL INLINE_ERROR_PROCESS       (interrupt$status);
END INTERRUPT_HANDLER;

INLINE_ERROR_PROCESS: PROCEDURE (INTERRUPT STATUS);
    IF interrupt$status <> E$OK THEN
        DO;
            •
            •  } In-line Error Processing PL/M-86 Statements
            •
        END;
END INLINE_ERROR_PROCESS;

SAMPLE_PROCEDURE:

    PROCEDURE;

    ds$pointer = @the$first$word; /* a dummy identifier used to point to
                                     interrupt handler's data segment */
    data$segment = ds$pointer$ovly.base;
                                  /* identifies the base address of the
                                     interrupt handler's data segment */
    intrpt$handlr$addrs = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                  /* points to the first instruction of
                                     the interrupt handler */
    interrupt$task$flag = 0;      /* indicates no interrupt task on level
                                     7 */
        •
        •  } Typical PL/M-86 Statements
        •

/*************************************************************************
 *  By first invoking the SET$INTERRUPT system call, the calling task   *
 *  sets up an interrupt level.                                          *
 ************************************************************************/

    CALL RQ$SET$INTERRUPT           (interrupt$level$7,
                                    interrupt$task$flag,
                                    interrupt$handler,
                                    data$segment,
                                    @status);
        •
        •  } Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	No value had previously been specified in the call to SET$INTERRUPT.
E$NOT$CON-FIGURED	This system call is not included in the present configuration.
E$PARAM	The level parameter is invalid.

EXIT$INTERRUPT

EXIT$INTERRUPT is used by interrupt handlers when they don't invoke interrupt tasks; this call sends an end-of-interrupt signal to the hardware.

```
CALL RQ$EXIT$INTERRUPT (level, except$ptr);
```

INPUT PARAMETER

level — A WORD specifying an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit of the interrupt level
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call. All exceptional conditions must be processed in-line, as control does not pass to an exception handler.

DESCRIPTION

The EXIT$INTERRUPT system call sends an end-of-interrupt signal to the hardware. This sets the stage for re-enabling interrupts. The re-enabling actually occurs when control passes from the interrupt handler to an application task.

EXAMPLE

```
/**********************************************************************
 *  This example illustrates how the EXIT$INTERRUPT system call can be  *
 *  used to send an end-of-interrupt signal to the hardware.            *
 **********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE interrupt$level$7       LITERALLY '0000 0000 0111 1000B';
                                    /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag     BYTE;
    DECLARE interrupt$handler       POINTER;
    DECLARE data$segment            WORD;
    DECLARE status                  WORD;
    DECLARE interrupt$status        WORD;

INTERRUPT_HANDLER: PROCEDURE INTERRUPT 59 PUBLIC;

          •
          •  }   Typical PL/M-86 Statements
          •

/**********************************************************************
 *  Interrupt handlers that do not invoke interrupt tasks need to      *
 *  invoke the EXIT$INTERRUPT system call to send an end-of-interrupt  *
 *  signal to the hardware.                                            *
 **********************************************************************/

    CALL RQ$EXIT$INTERRUPT          (interrupt$level$7,
                                    @interrupt$status);
    IF interrupt$status <> E$OK THEN
        DO;
            •
            •  }   In-line Error Processing PL/M-86 Statements
            •
        END;

END INTERRUPT_HANDLER;

SAMPLE_PROCEDURE:
    PROCEDURE;

    interrupt$task$flag = 0;        /* indicates no interrupt task on
                                       level 7 */
    data$segment = 0;               /* indicates that the interrupt handler
                                       will load its own data segment */
    interrupt$handler = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                    /* points to the first instruction of
                                       the interrupt handler */
          •
          •  }   Typical PL/M-86 Statements
          •
```

SYSTEM CALLS

```
/***********************************************************************
 *  By first invoking the SET$INTERRUPT system call, the calling task   *
 *  sets up an interrupt level.                                         *
 ***********************************************************************/

    CALL RQ$SET$INTERRUPT         (interrupt$level$7,
                                   interrupt$task$flag,
                                   interrupt$handler,
                                   data$segment,
                                   @status);
          •
          •   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The SET$INTERRUPT system call has not been invoked for the specified level.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

FORCE$DELETE

The FORCE$DELETE system call deletes objects whose disabling depths are zero or one.

CAUTION

DISABLE$DELETION makes an object immune to ordinary deletion by increasing the disabling depth of an object. If a Human Interface application contains objects whose disabling depths are greater than one, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal). Therefore you should not use DISABLE$DELETION (and have no need to use ENABLE$DELETION or FORCE$DELETE) in Human Interface applications.

```
CALL RQ$FORCE$DELETE(extension, object, except$ptr);
```

INPUT PARAMETERS

extension	If the object to be deleted is a composite object, this parameter is a TOKEN for the extension type associated with the composite object to be deleted. Otherwise, the extension parameter must be zero.
object	A TOKEN for the object that is to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The FORCE$DELETE system call deletes objects whose disabling depths are zero or one. If an object has a deletion depth of two or more, the calling task is put to sleep until the deletion depth is decreased to one. At that time, the object is deleted and the task is awakened.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	One or both of the object or extension paramenters does not refer to an existing object.
E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block for Nucleus housekeeping purposes.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	The extension parameter is a token for an ojbect that is not an extension type.

GET$EXCEPTION$HANDLER

GET$EXCEPTION$HANDLER returns information about the calling task's exception handler.

```
CALL RQ$GET$EXCEPTION$HANDLER (exception$info$ptr, except$ptr);
```

OUTPUT PARAMETERS

exception$info$ptr A POINTER to a structure of the following form:

```
STRUCTURE (
    EXCEPTION$HANDLER$OFFSET     WORD,
    EXCEPTION$HANDLER$BASE       WORD,
    EXCEPTION$MODE               BYTE);
```

where, after the call,

- exception$handler$offset contains the offset of the first instruction of the exception handler.
- exception$handler$base contains a base for the segment containing the first instruction of the exception handler. If exception$handler$base and exception$handler$offset are both zero, the calling task's exception handler is the system default exception handler.
- exception$mode contains an encoded indication of the calling task's current exception mode. The value is interpreted as follows:

Value	When to Pass Control to Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditons

except$ptr A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The GET$EXCEPTION$HANDLER system call returns both the address of the calling task's exception handler and the current value of the task's exception mode.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the GET$EXCEPTION$HANDLER system call   *
 *  can be used to return information about the calling task's          *
 *  exception handler.                                                   *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE x$handler  STRUCTURE    (x$handler$offset  WORD,
                                     x$handler$base    WORD,
                                     x$mode            BYTE);
    DECLARE status                   WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

          •
          •    Typical PL/M-86 Statements
          •

/**************************************************************************
 *  The address of the calling task's exception handler and the value    *
 *  of the task's exception mode (which specifies when to pass control   *
 *  to the exception handler) are both returned when the calling task    *
 *  invokes the GET$EXCEPTION$HANDLER system call.                       *
 **************************************************************************/

    CALL RQ$GET$EXCEPTION$HANDLER (@x$handler,
                                   @status);
          •
          •    Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

GET$LEVEL

GET$LEVEL returns the number of the level of the highest priority interrupt being serviced.

```
level = RQ$GET$LEVEL (except$ptr);
```

OUTPUT PARAMETERS

level — A WORD whose value is interpreted as follows (bit 15 is the high-order bit):

Bits	Value
15-8	reserved
7	if zero, some level is being serviced and bits 6-0 are significant if one, no level is being serviced and bits 6-0 are not significant
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

except$ptr — A POINTER to a WORD to which the condition code for the call is to be retured. All exceptional conditions must be processed in-line. Control does not pass to an exceptional handler.

DESCRIPTION

The GET$LEVEL system call returns to the calling task the highest (numerically lowest) level which an interrupt handler has started servicing but has not yet finished. To interpret the returned level number with more ease, strip away the unwanted bits (31-16) by logically ANDing the returned value with 00FFH.

EXAMPLE

```
/************************************************************************
 * This example illustrates how the GET$LEVEL system call can be used. *
 ************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE interrupt$level         WORD;
    DECLARE status                  WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

         •
         •  }    Typical PL/M-86 Statements
         •

/************************************************************************
 * The GET$LEVEL system call returns to the calling task the number of *
 * the highest interrupt level being serviced.                         *
 ************************************************************************/

    interrupt$level = RQ$GET$LEVEL (@status);
         •
         •  }    Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

GET$POOL$ATTRIB

GET$POOL$ATTRIB returns information about the memory pool of the calling task's job.

```
CALL RQ$GET$POOL$ATTRIB (attrib$ptr, except$ptr);
```

INPUT PARAMETER

attrib$ptr

A POINTER to a data structure of the following form:

```
STRUCTURE(
    POOL$MAX        WORD,
    POOL$MIN        WORD,
    INITIAL$SIZE    WORD,
    ALLOCATED       WORD,
    AVAILABLE       WORD);
```

where, after the call,

- POOL$MAX contains the maximum allowable size of the memory pool of the calling task's job.
- POOL$MIN contains the minimum allowable size of the memory pool of the calling task's job.
- INITIAL$SIZE contains the original value of the pool$min attribute.
- ALLOCATED contains the number of 16-byte paragraphs currently allocated from the memory pool of the calling task's job.
- AVAILABLE contains the number of 16-byte paragraphs currently available in the memory pool of the calling task's job. It does not include memory that could be borrowed from the parent job. The memory indicated in AVAILABLE may be fragmented and thus not allocatable as a single segment.

OUTPUT PARAMETER

except$ptr

A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The GET$POOL$ATTRIB system call returns information regarding the memory pool of the calling task's job. The data returned comprises the allocated and available portions of the pool, as well as its initial, minimum, and maximum sizes.

EXAMPLE

```
/**********************************************************************
 *  This example illustrates how the GET$POOL$ATTRIB system call can   *
 *  be use to return information about the memory pool of the the      *
 *  calling task's job.                                                *
 **********************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

   DECLARE mem$pool  STRUCTURE    (mem$pool$max      WORD,
                                  mem$pool$min       WORD,
                                  mem$initial$size   WORD,
                                  mem$allocated      WORD,
                                  mem$available      WORD);
   DECLARE status                 WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

        •
        • }   Typical PL/M-86 Statements
        •

/**********************************************************************
 *  The maximum and minimum size of the memory pool, the original value *
 *  of the minimum pool size, and the allocated and available number of *
 *  16-byte paragraphs in the memory pool of the calling task's job are *
 *  all returned when the calling task invokes the GET$POOL$ATTRIB      *
 *  system call.                                                        *
 **********************************************************************/

   CALL RQ$GET$POOL$ATTRIB        (@mem$pool,
                                  @status);
        •
        • }   Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

GET$PRIORITY

GET$PRIORITY returns the priority of a task.

```
priority = RQ$GET$PRIORITY (task, except$ptr);
```

INPUT PARAMETER

task	A TOKEN that specifies the task whose priority is being requested. • If not zero, the TOKEN contains a token for the task whose priority is being requested. • If zero, the calling task is asking for its own priority.

OUTPUT PARAMETERS

priority	A BYTE containing the priority of the task indicated by the task parameter.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The GET$PRIORITY system call returns the priority of the specified task.

EXAMPLE

```
/***********************************************************************
 *  This example illustrates how the GET$PRIORITY system call can be    *
 *  used.                                                               *
 ***********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);        /* Declares all system calls */

    DECLARE priority                 BYTE;
    DECLARE calling$tasks$priority   LITERALLY '0';
    DECLARE status                   WORD;
```

```
SAMPLE_PROCEDURE:
    PROCEDURE;

        •
        •  }   Typical PL/M-86 Statements
        •

/**************************************************************************
 *  The GET$PRIORITY system call returns the priority of the calling     *
 *  task.                                                                 *
 **************************************************************************/

    priority = RQ$GET$PRIORITY      (calling$tasks$priority,
                                    @status);
        •
        •  }   Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The task parameter is not a token for an existing object.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The task parameter is a token for an object that is not a task.

GET$SIZE

GET$SIZE returns the size, in bytes, of a segment.

```
size = RQ$GET$SIZE (segment, except$ptr);
```

INPUT PARAMETER

segment	A TOKEN for a segment.

OUTPUT PARAMETERS

size	A WORD that specifies the size of the segment. • If not zero, it contains the size, in bytes, of the segment indicated by the segment parameter. • If zero, the WORD indicates that the size of the segment is 65536 (64K) bytes.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The GET$SIZE system call returns the size, in bytes, of a segment.

EXAMPLE

```
/*************************************************************************
 * This example illustrates how the GET$SIZE system call can be used.  *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                   /* if your PL/M compiler does not
                                      support this variable type,
                                      declare TOKEN a WORD */
    DECLARE mbx$token              TOKEN;
    DECLARE calling$tasks$job      LITERALLY '0';
    DECLARE wait$forever           LITERALLY 'OFFFFH';
    DECLARE seg$token                TOKEN;
    DECLARE response                 TOKEN;
    DECLARE size                     WORD;
    DECLARE status                   WORD;

SAMPLE_PROCEDURE:

    PROCEDURE;

          •
          • }   Typical PL/M-86 Statements
          •

/*********************************************************************
 * In order to invoke the GET$SIZE system call, the calling task must *
 * know the token for the segment. In this example, the calling task *
 * invokes the LOOKUP$OBJECT and RECEIVE$MESSAGE system calls to     *
 * receive the token for a segment (seg$token). The calling task     *
 * invoked LOOKUP$OBJECT to receive the token for the mailbox named  *
 * 'MBX'. 'MBX' had been predesignated as the mailbox another task   *
 * would use to send an object.                                      *
 *********************************************************************/

    mbx$token = RQ$LOOKUP$OBJECT    (calling$tasks$job,
                                    @(3,'MBX'),
                                    wait$forever,
                                    @status);
          •
          • }   Typical PL/M-86 Statements
          •

/*********************************************************************
 * The RECEIVE$MESSAGE system call returns seg$token to the calling  *
 * task.                                                             *
 *********************************************************************/

    seg$token = RQ$RECEIVE$MESSAGE    (mbx$token,
                                      wait$forever,
                                      @response,
                                      @status);
          •
          • }   Typical PL/M-86 Statements
          •

/*********************************************************************
 * The GET$SIZE system call returns the size of the segment pointed  *
 * to by seg$token.                                                  *
 *********************************************************************/

    size = RQ$GET$SIZE                (seg$token,
                                      @status);
          •
          • }   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditons.
E$EXIST	The segment parameter is not a token for an existing object.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The segment parameter is a token for an object that is not a segment.

GET$TASK$TOKENS

GET$TASK$TOKENS returns the token requested by the calling task.

```
token = RQ$GET$TASK$TOKENS (selection, except$ptr);
```

INPUT PARAMETER

selection — A BYTE that tells the iRMX 86 Operating System what information is desired. It is encoded as follows:

Value	Object for which a Token is Requested
0	The calling task.
1	The calling task's job.
2	The parameter object of the calling task's job.
3	The root job.

OUTPUT PARAMETERS

token — A TOKEN to which the iRMX 86 Operating System will return the requested token.

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The GET$TASK$TOKENS system call returns a token for either the calling task, the calling task's job, the parameter object of the calling task's job, or the root job, depending on the encoded request.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the GET$TASK$TOKENS system call can be *
 *  used to return the TOKEN requested by the calling task.             *
 ************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);           /* Declares all system calls */

    DECLARE TOKEN                       LITERALLY 'SELECTOR';
                                        /* if your PL/M compiler does not
                                           support this variable type,
                                           declare TOKEN a WORD */
    DECLARE task$token                  WORD;
    DECLARE calling$task                LITERALLY '0';
    DECLARE status                      WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

          •
          •  }   Typical PL/M-86 Statements
          •

/************************************************************************
 *  If you set the selection parameter to zero, the GET$TASK$TOKENS     *
 *  system call will return a token for the calling task.               *
 ************************************************************************/

    task$token = RQ$GET$TASK$TOKENS     (calling$task,
                                        @status);
          •
          •  }   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$PARAM	The selection parameter is greater than 3.

GET$TYPE

GET$TYPE returns the encoded type of an object.

```
type$code = RQ$GET$TYPE (object, except$ptr);
```

INPUT PARAMETER

object — A TOKEN for an object.

OUTPUT PARAMETERS

type$code — A WORD which contains the encoded type of the specified object. The types for Nucleus objects are encoded as follows:

Value	Type
1	job
2	task
3	mailbox
4	semaphore
5	region
6	segment
7	extension
8	composite

Regions, extensions, and composites are described in Chapter 9.

except$ptr — A POINTER to a WORD to which the condition code for the call is returned.

DESCRIPTION

The GET$TYPE system call returns the type code for an object.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the GET$TYPE system call can be used    *
 *  to return the encoded type of an object.                              *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE type$code                  WORD;
    DECLARE mbx$token                  TOKEN;
    DECLARE calling$tasks$job          LITERALLY '0';
    DECLARE wait$forever               LITERALLY '0FFFFH';
    DECLARE object$token               TOKEN;
    DECLARE response                   TOKEN;
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:

    PROCEDURE;

        •
        • }   Typical PL/M-86 Statements
        •

/**************************************************************************
 *  In order to invoke the GET$TYPE system call, the calling task must   *
 *  have the token for an object.  In this example, the calling task     *
 *  invokes the LOOKUP$OBJECT system call and then the RECEIVE$MESSAGE   *
 *  system call to receive the token for an object of unknown type       *
 *  (object$token).                                                       *
 **************************************************************************/

    mbx$token = RQ$LOOKUP$OBJECT       (calling$tasks$job,
                                       @(3,'MBX'),
                                       wait$forever,
                                       @status);

        •
        • }   Typical PL/M-86 Statements
        •

/**************************************************************************
 *  The RECEIVE$MESSAGE system call returns object$token to the calling *
 *  task after the calling task invoked LOOKUP$OBJECT to receive the     *
 *  token for the mailbox named 'MBX'.  'MBX' had been predesignated     *
 *  as the mailbox another task would use to send an object.             *
 **************************************************************************/

    object$token = RQ$RECEIVE$MESSAGE  (mbx$token,
                                       wait$forever,
                                       @response,
                                       @status);
        .
        . }  Typical PL/M-86 Statements
        .

/*********************************************************************
 *  Using the type code returned by the GET$TYPE system call, the      *
 *  calling task can find out if the object is a job, a task, a        *
 *  mailbox, a region, or a segment.                                   *
 *********************************************************************/

    type$code = RQ$GET$TYPE            (object$token,
                                        @status);

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The object parameter is not a token for an existing object.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

INSPECT$COMPOSITE

The INSPECT$COMPOSITE system call returns a list of the component tokens contained in a composite object.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal).

```
CALL RQ$INSPECT$COMPOSITE(extension, composite, token$list,
                          except$ptr);
```

INPUT PARAMETERS

extension	A TOKEN for the extension object corresponding to the composite object being inspected.
composite	A TOKEN for the composite object being inspected.

OUTPUT PARAMETERS

token$list A POINTER to a structure of the form:

```
Declare
   token$list   STRUCTURE(
   num$slots       WORD,
   num$used        WORD,
   tokens(*)       TOKEN);
```

where:

num$slots	Number of positions available for tokens in token$list (an upper limit on the number of tokens to be returned).

num$used	Number of component tokens making up the composite object.
token(*)	The tokens that actually constitute the composite object.

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The INSPECT$COMPOSITE system call accepts a token for a composite object and returns a list of tokens for the components of the composite object.

The calling task must supply the num$slots value in the data structure pointed to by the token$list parameter. The Nucleus fills in the remaining fields in that structure. If num$slots is set to zero, the Nucleus will fill in only the num$used field.

If the num$slots value is smaller than the actual number of component tokens, only that number (num$slots) of tokens will be returned.

EXAMPLE

See the example in section "DELETE_RING_BUFFER Procedure" of Chapter 11.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The composite parameter is not compatible with the extension parameter.
E$EXIST	One or both of the extension or composite parameters does not refer to a currently existing object.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	One or both of the extension or composite parameters is not of the correct object type.

LOOKUP$OBJECT

LOOKUP$OBJECT returns a token for a cataloged object.

```
object = RQ$LOOKUP$OBJECT (job, name, time$limit, except$ptr);
```

INPUT PARAMETERS

job	A TOKEN indicating the object directory to be searched. • If not zero, the TOKEN contains a token for the job whose object directory is to be searched. • If zero, the object directory to be searched is that of the calling task's job.
name	A POINTER to a STRING which contains the name under which the object is cataloged. During the lookup operation, upper and lower case letters are treated as being different.
time$limit	A WORD indicating the task's willingness to wait. • If zero, the WORD indicates that the calling task is not willing to wait. • If OFFFFH, the WORD indicates that the task will wait as long as is necessary. • If between 0 and OFFFFH, the WORD indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is a configuration option. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

OUTPUT PARAMETERS

object	A TOKEN containing the requested token.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The LOOKUP$OBJECT system call returns the token for the specified object after searching for its name in the specified object directory. Because it is possible that the object is not cataloged at the time of the call, the calling task has the option of waiting, either indefinitely or for a specific period of time, for another task to catalog the object.

EXAMPLE

```
/*************************************************************************
 *  This example illustrates how the LOOKUP$OBJECT system call can be   *
 *  used to return a token for a cataloged object.                      *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE mbx$token                  TOKEN;
    DECLARE calling$tasks$job          LITERALLY '0';
    DECLARE wait$forever               LITERALLY 'OFFFFH';
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;
         •
         •  }   Typical PL/M-86 Statements
         •

/*************************************************************************
 *  In this example, the calling task invokes LOOKUP$OBJECT in order to *
 *  search the object directory of the calling task's job for an object *
 *  with the name 'MBX'.                                                 *
 *************************************************************************/

    mbx$token = RQ$LOOKUP$OBJECT       (calling$tasks$job,
                                       @(3,'MBX'),
                                       wait$forever,
                                       @status);
         •
         •  }   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The specified job has an object directory of size 0.
E$EXIST	One of the following is true: • The specified job was deleted while the task was waiting. • The job parameter (which is not zero) is not a token for an existing object. • The name was found, but the cataloged object has a null (zero) token.
E$LIMIT	The specified object directory is full and the object being looked-up has not yet been cataloged.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The first byte of the string pointed to by the name parameter contains a value greater than 12 or equal to zero.
E$TIME	One of the following is true: • The calling task indicated its willingness to wait a certain amount of time, then waited without satisfaction. • The task was not willing to wait, and the entry indicated by the name parameter is not in the specified object directory.
E$TYPE	The job parameter contains a token for an object that is not a job.

OFFSPRING

OFFSPRING returns a token for each child (job) of a job.

```
token$list = RQ$OFFSPRING (job, except$ptr);
```

INPUT PARAMETER

job	A TOKEN for the job whose offspring are desired. A value of zero specifies the calling task's job.

OUTPUT PARAMETER

token$list — A TOKEN that indicates the children of the specified job.

- If not zero, the TOKEN contains a token for a segment. The first word in the segment contains the number of words in the remainder of the segment. Subsequent words contain the tokens for jobs which are the immediate children of the specified job.
- If zero, the specified job has no children.

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The OFFSPRING system call returns the token for a segment. The segment contains a token for each child of the specified job. By repeated use of this call, tokens can be obtained for all descendents of a job; this information is needed by a task which is attempting to delete a job that has child jobs.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the OFFSPRING system call can be used  *
 *  to return a token for each child of a job.                          *
 ************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE token$list                 TOKEN;
    DECLARE calling$tasks$job          LITERALLY '0';
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

         •
         • }   Typical PL/M-86 Statements
         •

/************************************************************************
 *  In this example, the calling task invokes the system call OFFSPRING *
 *  to obtain a token for a segment.  This segment contains the tokens  *
 *  for jobs that are immediate children of the calling task's job.     *
 ************************************************************************/

    token$list = RQ$OFFSPRING          (calling$tasks$job,
                                       @status);
         •
         • }   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The job parameter does not contain a token for an existing object.
E$LIMIT	The required segment, if allocated, would exceed the job object limit.
E$MEM	There is not sufficient memory available to create the required segment.

E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The job parameter contains a token for an object that is not a job.

RECEIVE$CONTROL

The RECEIVE$CONTROL system call allows the calling task to gain access to data protected by a region.

CAUTION

Tasks which use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal) while the task is in the region.

```
CALL RQ$RECEIVE$CONTROL (region, except$ptr);
```

INPUT PARAMETER

region	A TOKEN for the region protecting the data to which the calling task wants access.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The RECEIVE$CONTROL system call requests access to data protected by a region. If no task currently has access, entry is immediate. If another task currently has access, the calling task is placed in the region's task queue and goes to sleep. The task remains asleep until it gains access to the data.

If the region has a priority-based task queue, the priority of the task currently having access is temporarily boosted, if necessary, to match that of the task at the head of the queue.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the RECEIVE$CONTROL system call can be *
 *  used to gain access to data protected by a region.                  *
 ************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

   DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                      /* if your PL/M compiler does not
                                         support this variable type,
                                         declare TOKEN a WORD */
   DECLARE region$token               TOKEN;
   DECLARE priority$queue             LITERALLY '1'; /* tasks wait in
                                                        priority order */
   DECLARE status                     WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

         •
         •  }   Typical PL/M-86 Statements
         •

/************************************************************************
 *  In order to access the data within a region, a task must know the   *
 *  token for that region.  In this example, the needed token is known  *
 *  because the calling task creates the region.                         *
 ************************************************************************/

   region$token = RQ$CREATE$REGION    (priority$queue,
                                      @status);
         •
         •  }   Typical PL/M-86 Statements
         •

/************************************************************************
 *  When access to the data protected by a region is needed, the        *
 *  calling task may invoke the RECEIVE$CONTROL system call.            *
 ************************************************************************/

   CALL RQ$RECEIVE$CONTROL            (region$token,
                                      @status);
         •
         •  }   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The region parameter refers to a region already accessed by the calling task.
E$EXIST	The region parameter does not contain a token for an existing object.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	The region parameter contains a token for an object that is not a region.

RECEIVE$MESSAGE

RECEIVE$MESSAGE delivers the calling task to a mailbox, where it can wait for an object token to be returned.

```
object = RQ$RECEIVE$MESSAGE (mailbox, time$limit, response$ptr,
                             except$ptr);
```

INPUT PARAMETERS

mailbox	A TOKEN for the mailbox at which the calling task expects to receive an object token.
time$limit	A WORD that indicates how long the calling task is willing to wait.

- if zero, indicates that the calling task is not willing to wait.
- if 0FFFFH, indicates that the task will wait as long as is necessary.
- if between 0 and 0FFFFH, indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is configurable. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

OUTPUT PARAMETERS

object	A TOKEN for the object being received.
response$ptr	A POINTER to a WORD in which the system returns a value. The returned word,

- if not zero, contains a token for the exchange to which the receiving task is to send a response.
- if zero, indicates that no response is expected by the sending task.

CAUTION

Response$ptr points to a location for the sending task to use. If you specify a constant value for response$ptr, be careful to ensure that the value does not conflict with system requirements.

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The RECEIVE$MESSAGE system call causes the calling task either to get the token for an object or to wait for the token in the task queue of the specified mailbox. If the object queue at the mailbox is not empty, then the calling task immediately gets the token at the head of the queue and remains ready. Otherwise, the calling task goes into the task queue of the mailbox and goes to sleep, unless the task is not willing to wait. In the latter case, or if the task's waiting period elapses without a token arriving, the task is awakened with an E$TIME exceptional condition.

It is possible that the token returned by RECEIVE$MESSAGE is a token for an object that has already been deleted. To verify that the token is valid, the receiving task can invoke the GET$TYPE system call. However, tasks can avoid this situation by adhering to proper programming practices.

One such practice is for the sending task to request a response from the receiving task and not delete the object until it gets a response. When the receiving task finishes with the object, it sends a response, the nature of which must be determined by the writers of the two tasks, to the response mailbox. When the sending task gets this response, it can then delete the original object, if it so desires.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the RECEIVE$MESSAGE system call can be *
 *  used to receive a message segment.                                   *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE mbx$token                  TOKEN;
    DECLARE calling$tasks$job          LITERALLY '0';
    DECLARE wait$forever               LITERALLY '0FFFFH';
    DECLARE seg$token                  TOKEN;
    DECLARE response                   TOKEN;
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

         •
         •  }    Typical PL/M-86 Statements
         •

/**************************************************************************
 *  In this example the calling task looks up the token for the mailbox *
 *  prior to invoking the RECEIVE$MESSAGE system call.                   *
 **************************************************************************/

    mbx$token = RQ$LOOKUP$OBJECT       (calling$tasks$job,
                                       @(3,'MBX'),
                                       wait$forever,
                                       @status);

         •
         •  }    Typical PL/M-86 Statements
         •

/**************************************************************************
 *  Knowing the token for the mailbox, the calling task can wait for a  *
 *  message from this mailbox by invoking the RECEIVE$MESSAGE system    *
 *  call.                                                                *
 **************************************************************************/

    seg$token = RQ$RECEIVE$MESSAGE     (mbx$token,
                                       wait$forever,
                                       @response,
                                       @status);

         •
         •  }    Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	One of the following is true: • The mailbox parameter does not contain a token for an existing object. • The mailbox was deleted while the task was waiting.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TIME	One of the following is true: • The calling task was not willing to wait and there was not a token available. • The task waited in the task queue and its designated waiting period elapsed before the task got the desired token.
E$TYPE	The mailbox parameter contains a token for an object that is not a mailbox.

RECEIVE$UNITS

RECEIVE$UNITS delivers the calling task to a semaphore, where it waits for units.

```
value = RQ$RECEIVE$UNITS (semaphore, units, time$limit, except$ptr);
```

INPUT PARAMETERS

semaphore	A TOKEN for the semaphore from which the calling task hopes to receive units.
units	A WORD containing the number of units that the calling task is requesting.
time$limit	A WORD that indicates how long the calling task is willing to wait.

- If zero, the WORD indicates that the calling task is not willing to wait.
- If OFFFFH, the WORD indicates that the task will wait as long as is necessary.
- If between 0 and OFFFFH, the WORD indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is configurable. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

OUTPUT PARAMETERS

value	A WORD containing the number of units remaining in the custody of the semaphore after the calling task's request is satisfied.
except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The RECEIVE$UNITS system call causes the calling task either to get the units that it is requesting or to wait for them in the semaphore's task queue. If the units are available and the task is at the front of the queue, then the task receives them and remains ready. Otherwise, the task is placed in the semaphore's task queue and goes to sleep, unless the task is not willing to wait. In the latter case, or if the task's waiting period elapses before the requested units are available, the task is awakened with an E$TIME exceptional condition.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the RECEIVE$UNITS system call can be    *
 *  used to receive a unit.                                               *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE sem$token                  TOKEN;
    DECLARE calling$tasks$job          LITERALLY '0';
    DECLARE wait$forever               LITERALLY 'OFFFFH';
    DECLARE seg$token                  TOKEN;
    DECLARE units$remaining            WORD;
    DECLARE units$requested            WORD;
    DECLARE status                     WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

            •
            •   Typical PL/M-86 Statements
            •

/**************************************************************************
 *  In this example the calling task looks up the token for the          *
 *  semaphore prior to invoking the RECEIVE$UNITS system call.           *
 **************************************************************************/

    sem$token = RQ$LOOKUP$OBJECT       (calling$tasks$job,
                                       @(5,'SEMA4'),
                                       wait$forever,
                                       @status);

            •
            •   Typical PL/M-86 Statements
            •
```

SYSTEM CALLS

```
/************************************************************************
 *  Knowing the token for the semaphore, the calling task can wait for  *
 *  units at this semaphore by invoking the RECEIVE$UNITS system call.  *
 ************************************************************************/

    units$remaining = RQ$RECEIVE$UNITS (sem$token,
                                        units$requested,
                                        wait$forever,
                                        @status);
          .
          .  }   Typical PL/M-86 Statements
          .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	One of the following is true: • The semaphore parameter is not a token for an existing object. • The semaphore was deleted while the task was waiting.
E$LIMIT	The units parameter is greater than the maximum value that had been specified for the semaphore when it was created.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TIME	One of the following is true: • The calling task was not willing to wait and the requested units were not available. • The task waited in the task queue and its designated waiting period elapsed before the requested units were available.
E$TYPE	The semaphore parameter is a token for an object that is not a semaphore.

RESET$INTERRUPT

RESET$INTERRUPT cancels the assignment of an interrupt handler to a level.

```
   CALL RQ$RESET$INTERRUPT (level, except$ptr);
```

INPUT PARAMETER

level — A WORD specifying an interrupt level which is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The RESET$INTERRUPT system call cancels the assignment of the current interrupt handler to the specified interrupt level. If an interrupt task had also been assigned to the level, the interrupt task is deleted. RESET$INTERRUPT also disables the level.

The level reserved for the system clock should not be reset and is considered invalid. This level is a configuration option (refer to the iRMX 86 CONFIGURATION GUIDE for further information).

EXAMPLE

```
/*************************************************************************
 * This example illustrates how the RESET$INTERRUPT system call can be *
 * used to cancel the assignment of an interrupt handler to an         *
 * interrupt level.                                                    *
 *************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                   /* if your PL/M compiler does not
                                      support this variable type,
                                      declare TOKEN a WORD */
    DECLARE task$token             TOKEN;
    DECLARE priority$level$66      LITERALLY '66';
    DECLARE start$address          POINTER;
    DECLARE data$segment           WORD;
    DECLARE stack$pointer          POINTER;
    DECLARE stack$size$512         LITERALLY '512'; /* new task's stack
                                                       size is 512 bytes */
    DECLARE task$flags             WORD;
    DECLARE interrupt$level$7      LITERALLY '0000 0000 0111 1000B';
                                   /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag    BYTE;
    DECLARE intrpt$handlr$addrs    POINTER;
    DECLARE interrupt$status       WORD;
    DECLARE status                 WORD;

INTERRUPT_TASK: PROCEDURE PUBLIC;

    interrupt$task$flag = 001H;    /* indicates that calling task is to
                                      be interrupt task */
    data$segment = 0;              /* use own data segment */
    intrpt$handlr$addrs = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                   /* points to the first instruction of
                                      the interrupt handler */

/*************************************************************************
 * The first system call in this example, SET$INTERRUPT, makes the     *
 * calling task (INTERRUPT_TASK) the interrupt task for the interrupt  *
 * level.                                                              *
 *************************************************************************/

    CALL RQ$SET$INTERRUPT          (interrupt$level$7,
                                   interrupt$task$flag,
                                   intrpt$handlr$addrs,
                                   data$segment,
                                   @interrupt$status);
```

```
/************************************************************************
 *  The second system call, WAIT$INTERRUPT, is used by the interrupt    *
 *  task to signal its readiness to service an interrupt.               *
 ************************************************************************/

    CALL RQ$WAIT$INTERRUPT        (interrupt$level$7,
                                   @interrupt$status);
         •
         • }   Typical PL/M-86 Statements
         •

/************************************************************************
 *  When the interrupt task invokes the RESET$INTERRUPT system call,    *
 *  the assignment of the current interrupt handler to interrupt level  *
 *  7 is canceled and, because an interrupt task has also been assigned *
 *  to the level, the interrupt task is deleted.                        *
 ************************************************************************/

    CALL RQ$RESET$INTERRUPT       (interrupt$level$7,
                                   @interrupt$status);
END INTERRUPT_TASK;

SAMPLE_PROCEDURE:
    PROCEDURE;

    start$address = @INTERRUPT_TASK;
                                  /* 1st instruction of interrupt task */
    stack$pointer = 0;            /* automatic stack allocation  */
    task$flags = 0;               /* indicates no floating-point
                                     instructions */
    data$segment = 0;             /* use own data segment */
         •
         • }   Typical PL/M-86 Statements
         •

/************************************************************************
 *  In this example the SAMPLE_PROCEDURE is needed to create the task   *
 *  labeled INTERRUPT_TASK.                                             *
 ************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$66,
                                   start$address,
                                   data$segment,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);
         •
         • }   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	There is not an interrupt handler assigned to the specified level.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

RESUME$TASK

RESUME$TASK decreases by one the suspension depth of a task.

```
CALL RQ$RESUME$TASK (task, except$ptr);
```

INPUT PARAMETER

task	A TOKEN for the task whose suspension depth is to be decremented.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The RESUME$TASK system call decreases by one the suspension depth of the specified non-interrupt task. The task should be in either the suspended or asleep-suspended state, so its suspension depth should be at least one. If the suspension depth is still positive after being decremented, the state of the task is not changed. If the depth becomes zero, and the task is in the suspended state, then it is placed in the ready state. If the depth becomes zero, and the task is in the asleep-suspended state, then it is placed in the asleep state.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the RESUME$TASK system call can be      *
 *  used to decrease by one the suspension depth of a task.              *
 **************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

   TASK_CODE: PROCEDURE EXTERNAL;
   END TASK_CODE;
```

```
    DECLARE TOKEN                 LITERALLY 'SELECTOR';
                                  /* if your PL/M compiler does not
                                     support this variable type,
                                     declare TOKEN a WORD */
    DECLARE task$token            TOKEN;
    DECLARE priority$level$200    LITERALLY '200';
    DECLARE start$address         POINTER;
    DECLARE data$seg              WORD;
    DECLARE stack$pointer         POINTER;
    DECLARE stack$size$512        LITERALLY '512'; /* new task's stack
                                                      size is 512 bytes */
    DECLARE task$flags            WORD;
    DECLARE status                WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    start$address = @TASK_CODE;   /* first instruction of the new task */
    data$seg = 0;                 /* task sets up own data seg */
    stack$pointer = 0;            /* automatic stack allocation  */
    task$flags = 0;               /* indicates no floating-point
                                     instructions */
          •
          •  }   Typical PL/M-86 Statements
          •

/**************************************************************************
 *  In this example the calling task creates a non-interrupt task and    *
 *  suspends that task before invoking the RESUME$TASK system call.      *
 **************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$200,
                                  start$address,
                                  data$seg,
                                  stack$pointer,
                                  stack$size$512,
                                  task$flags,
                                  @status);
          •
          •  }   Typical PL/M-86 Statements
          •

/**************************************************************************
 *  After creating the task, the calling task invokes SUSPEND$TASK.      *
 *  This system call increases by one the suspension depth of the new    *
 *  task (whose code is labeled TASK_CODE).                              *
 **************************************************************************/

    CALL RQ$SUSPEND$TASK          (task$token,
                                  @status);
          •
          •  }   Typical PL/M-86 Statements
          •
```

```
/*************************************************************************
 *  Using the token for the suspended task (whose code is labeled       *
 *  TASK_CODE), the calling task invokes RESUME$TASK to decrease by the *
 *  one the suspension depth of the suspended task.                     *
 *************************************************************************/

    CALL RQ$RESUME$TASK          (task$token,
                                  @status);
          •
          •   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The task indicated by the task parameter is an interrupt task.
E$EXIST	The task parameter is not a token for an existing object.
E$STATE	The task indicated by the task parameter was not suspended when the call was made.
E$TYPE	The task parameter is a token for an object that is not a task.

SEND$CONTROL

The SEND$CONTROL system call allows a task to surrender access to data protected by a region.

CAUTION

Tasks which use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CTRL/c entered at a terminal) while the task is in the region.

```
CALL RQ$SEND$CONTROL (except$ptr);
```

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

When a task finishes with data protected by a region, the task invokes the SEND$CONTROL system call to surrender access. If the task is using more than one set of data, each of which is protected by a region, the SEND$CONTROL system call surrenders the most recently obtained access. When access is surrendered, the system allows the next task in line to gain access.

If a task calling SEND$CONTROL has had its priority boosted while it had access through a region, its priority is restored when it relinquishes the access.

EXAMPLE

```
/***********************************************************************
 *  This example illustrates how the SEND$CONTROL system call can be   *
 *  used to surrender access to data protected by a region.            *
 ***********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    DECLARE TOKEN                      LITERALLY 'SELECTOR';
                                       /* if your PL/M compiler does not
                                          support this variable type,
                                          declare TOKEN a WORD */
    DECLARE region$token               TOKEN;
    DECLARE priority$queue             LITERALLY '1'; /* tasks wait in
                                                         priority order */
    DECLARE status                     WORD;
          •
          •   Typical PL/M-86 Statements
          •

SAMPLE_PROCEDURE:
    PROCEDURE;

/***********************************************************************
 *  In order to access the data within a region, a task must know the  *
 *  token for that region.  In this example, the needed token is known *
 *  because the calling task creates the region.                       *
 ***********************************************************************/
    region$token = RQ$CREATE$REGION    (priority$queue,
                                       @status);
          •
          •   Typical PL/M-86 Statements
          •

/***********************************************************************
 *  When access to the data protected by a region is needed, the       *
 *  calling task may invoke the RECEIVE$CONTROL system call.           *
 ***********************************************************************/
    CALL RQ$RECEIVE$CONTROL            (region$token,
                                       @status);
          •
          •   Typical PL/M-86 Statements
          •

/***********************************************************************
 *  When a task finishes using data protected by a region, the task    *
 *  invokes the SEND$CONTROL system call to surrender access.          *
 ***********************************************************************/

    CALL RQ$SEND$CONTROL               (@status);
          •
          •   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	A task invoked the SEND$CONTROL system call while it did not have access to data protected by any region.
E$NOT$CONFIGURED	This system call is not part of the present configuration.

SEND$MESSAGE

SEND$MESSAGE sends an object token to a mailbox.

```
CALL RQ$SEND$MESSAGE (mailbox, object, response, except$ptr);
```

INPUT PARAMETERS

mailbox	A TOKEN for the mailbox to which an object token is to be sent.
object	A TOKEN containing an object token which is to be sent.
response	A TOKEN for a mailbox or semaphore at which the sending task will wait for a response. • if not zero, contains a token for the desired response mailbox or semaphore. • if zero, indicates that no response is requested.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SEND$MESSAGE system call sends the specified object token to the specified mailbox. If there are tasks in the task queue at that mailbox, the task at the head of the queue is awakened and is given the token. Otherwise, the object token is placed at the tail of the object queue of the mailbox. The sending task has the option of specifying a mailbox or semaphore at which it will wait for a response from the task that receives the object. The nature of the response must be agreed upon by the writers of the two tasks.

EXAMPLE

```
/**********************************************************************
 *  This example illustrates how the SEND$MESSAGE system call can be    *
 *  used to send a segment token to a mailbox.                          *
 **********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    DECLARE TOKEN                  LITERALLY 'SELECTOR';
                                   /* if your PL/M compiler does not
                                      support this variable type,
                                      declare TOKEN a WORD */
    DECLARE seg$token              TOKEN;
    DECLARE size                   WORD;
    DECLARE mbx$token              TOKEN;
    DECLARE mbx$flags              WORD;
    DECLARE no$response            LITERALLY '0';
    DECLARE status                 WORD;
    DECLARE job                    WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    size = 64;                     /* designates new segment to contain 64
                                      bytes */
    mbx$flags = 0;                 /* designates four objects to be queued
                                      on the high performance object
                                      queue;  designates a first-in/
                                      first-out task queue. */
    job = 0;                       /* indicates objects to be cataloged
                                      into the object directory of the
                                      calling task's job */

          .
          .    Typical PL/M-86 Statements
          .

/**********************************************************************
 *  The calling task creates a segment and a mailbox and catalogs the   *
 *  mailbox token.  The calling task then uses the tokens for both      *
 *  objects to send a message.                                          *
 **********************************************************************/

    seg$token = RQ$CREATE$SEGMENT  (size,
                                   @status);
    mbx$token = RQ$CREATE$MAILBOX  (mbx$flags,
                                   @status);
```

```
/************************************************************************
 *  It is not mandatory for the calling task to catalog the mailbox     *
 *  token in order to send a message.  It is necessary, however, to     *
 *  catalog (or in someway communicate) the mailbox token if another    *
 *  task is to receive the message.                                     *
 ************************************************************************/

    CALL RQ$CATALOG$OBJECT          (job,
                                    mbx$token,
                                    @(3, 'MBX'),
                                    @status);

          •
          •  }   Typical PL/M-86 Statements
          •

/************************************************************************
 *  The calling task invokes the SEND$MESSAGE system call to send the   *
 *  token for the segment to the specified mailbox.                     *
 ************************************************************************/

    CALL RQ$SEND$MESSAGE            (mbx$token,
                                    seg$token,
                                    no$response,
                                    @status);
          •
          •  }   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	One or more of the input parameters is not a token for an existing object.
E$MEM	The high performance queue is full and there is not sufficient memory in the job containing the mailbox for the Nucleus to do the housekeeping that supports a send message operation.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	One of the following is true: • The mailbox parameter is a token for an object that is not a mailbox. • The response parameter is a token for an object that is neither a mailbox nor a semaphore.

SEND$UNITS

SEND$UNITS sends units to a semaphore.

```
CALL RQ$SEND$UNITS (semaphore, units, except$ptr);
```

INPUT PARAMETERS

semaphore	A TOKEN for the semaphore to which the units are to be sent.
units	A WORD containing the number of units to be sent.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SEND$UNITS system call sends the specified number of units to the specified semaphore. If the transmission would cause the semaphore's supply of units to exceed its maximum allowawble supply, then an E$LIMIT exceptional condition occurs. Otherwise, the transmission is successful and the Nucleus attempts to satisfy the requests of the tasks in the semaphore's task queue, beginning at the head of the queue.

EXAMPLE

```
/***********************************************************************
 *  This example illustrates how the SEND$UNITS system call can be used *
 *  to send units to a semaphore.                                       *
 ***********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE TOKEN                   LITERALLY 'SELECTOR';
                                    /* if your PL/M compiler does not
                                       support this variable type,
                                       declare TOKEN a WORD */
    DECLARE sem$token              TOKEN;
    DECLARE init$value             WORD;
    DECLARE max$value              WORD;
    DECLARE sem$flags              WORD;
    DECLARE three$units$sent       LITERALLY '3';
    DECLARE status                 WORD;
    DECLARE job                    WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    init$value = 1;                /* the new semaphore has one initial
                                      unit */
    max$value = 10H;               /* the new semaphore can have a maximum
                                      of 16 units */
    sem$flags = 0;                 /* designates a first-in/
                                      first-out task queue. */
    job = 0;                       /* indicates objects to be cataloged
                                      into the object directory of the
                                      calling task's job */

            •
            •  }    Typical PL/M-86 Statements
            •

/*************************************************************************
 *  The calling task creates a semaphore and catalogs the semaphore      *
 *  token.  The calling task then uses the token to send a unit.         *
 *************************************************************************/

    sem$token = RQ$CREATE$SEMAPHORE  (init$value,
                                      max$value,
                                      sem$flags,
                                      @status);

            •
            •  }    Typical PL/M-86 Statements
            •

/*************************************************************************
 *  It is not mandatory to catalog the semaphore token in order to send *
 *  units.  It is necessary, however, to catalog (or in someway          *
 *  communicate) the semaphore token if another task is to receive the  *
 *  units.                                                               *
 *************************************************************************/

    CALL RQ$CATALOG$OBJECT         (job,
                                    sem$token,
                                    @(5, 'SEMA4'),
                                    @status);

            •
            •  }    Typical PL/M-86 Statements
            •
```

```
/*************************************************************************
 *  The calling task invokes the SEND$UNITS system call to send the     *
 *  units to the semaphore just created (sem$token.)                    *
 *************************************************************************/

    CALL RQ$SEND$UNITS              (sem$token,
                                    three$units$sent,
                                    @status);
          •
          •   }   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditons.
E$EXIST	The semaphore parameter is not a token for an existing object.
E$LIMIT	The number of units that the calling task is trying to send would cause the semaphore's supply of units to exceed its maximum allowable supply.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$TYPE	The semaphore parameter is a token for an object that is not a semaphore.

SET$EXCEPTION$HANDLER

SET$EXCEPTION$HANDLER assigns an exception handler to the calling task.

```
CALL RQ$SET$EXCEPTION$HANDLER (exception$info$ptr, except$ptr);
```

INPUT PARAMETER

exception$info$ptr A POINTER to a structure of the following form:

```
STRUCTURE(
   EXCEPTION$HANDLER$OFFSET    WORD,
   EXCEPTION$HANDLER$BASE      WORD,
   EXCEPTION$MODE              BYTE);
```

where:

- exception$handler$offset contains the offset of the first instruction of the exception handler.
- exception$handler$base contains the base of the iAPX 86 segment containing the first instruction of the exception handler.
- exception$mode contains an encoded indication of the calling task's intended exception mode. The value is interpreted as follows:

Value	When to Pass Control To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

If exception$handler$offset and exception$handler$base both contain zeros, the exception handler of the calling task's parent job is assigned.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SET$EXCEPTION$HANDLER system call enables a task to set its exception handler and exception mode attributes. If you want to designate the Debugger as the exception handler to interactively examine system objects and lists, the following code sets up the needed structure in PL/M-86:

```
    DECLARE    X    STRUCTURE (OFFSET    WORD,
                               BASE      WORD,
                               MODE      BYTE); /* establish a structure for
                                                   exception handlers */
    DECLARE    Y    POINTER AT (@X);

    DECLARE EXCEPTION WORD;

    Y = @RQDEBUGGEREX;                          /* designate the debugger
                                                   as the exception handler*/
    X.MODE = ZERO$ONE$TWO$OR$THREE;             /* the mode is a value 0-3 */
    CALL  RQ$SET$EXCEPTION$HANDLER (@X, @EXCEPTION);
```

EXAMPLE

```
/***********************************************************************
 *  This example illustrates how the SET$EXCEPTION$HANDLER system call  *
 *  can be used to assign an exception handler to the calling task.     *
 ***********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);          /* Declares all system calls */

    EXCEPTION_HANDLER: PROCEDURE EXTERNAL;
    END EXCEPTION_HANDLER;

    DECLARE X$HANDLER$STRUCTURE        LITERALLY 'STRUCTURE  offset  WORD,
                                                             base    WORD,
                                                             mode    BYTE)';
                                       /* establishes a structure for
                                          exception handlers */

    DECLARE x$handler                  X$HANDLER$STRUCTURE;
                                       /* using the exception handler
                                          structure, the pointer to the
                                          old exception handler is
                                          defined */
    DECLARE new$x$handler              X$HANDLER$STRUCTURE;
                                       /* using the exception handler
                                          structure, the new exception
                                          handler is defined */
    DECLARE all$exceptions        LITERALLY '3';
                                  /* control is passed to the exception
                                     handler on all exceptional
                                     conditions */
    DECLARE PTR$OVERLAY           LITERALLY 'STRUCTURE offset  WORD,
                                                       base    WORD)';
                                  /* establishes a structure for
                                     overlays */
    DECLARE seg$pointer           POINTER;
    DECLARE seg$pointer$ovly      PTR$OVERLAY AT (@seg$pointer);
                                  /* using the overlay structure, the
                                     first instruction of the
                                     exception handler is identified */
    DECLARE status                WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;
seg$pointer = @EXCEPTION_HANDLER; /* pointer to exception handler */
new$x$handler.offset = seg$pointer$ovly.offset;
                                  /* offset of the first instruction
                                     of the exception handler */
new$x$handler.base = seg$pointer$ovly.base;
                                  /* base address of the exception
                                     handler 8086 segment containing
                                     the first instruction of the
                                     exception handler */
new$x$handler.mode = all$exceptions;
                                  /* pass control on all conditions */
          •
          •  }    Typical PL/M-86 Statements
          •

/**************************************************************************
 *  The address of the calling task's exception handler and the value    *
 *  of the task's exception mode (when to pass control to the exception *
 *  handler) are both returned when the calling task invokes the         *
 *  GET$EXCEPTION$HANDLER system call.                                   *
 **************************************************************************/

    CALL RQ$GET$EXCEPTION$HANDLER     (@x$handler,
                                       @status);
          •
          •  }    Typical PL/M-86 Statements
          •
```

```
/************************************************************************
 *  The calling task may invoke the SET$EXCEPTION$HANDLER system call   *
 *  to first set a new exception handler and then to later reset the    *
 *  old exception handler.                                              *
 ************************************************************************/

    CALL RQ$SET$EXCEPTION$HANDLER       (@new$x$handler,
                                         @status);
          •
          •  }    Typical PL/M-86 Statements
          •

/************************************************************************
 *  No longer needing the new exception handler, the calling task uses  *
 *  the address and mode of the old exception handler to return         *
 *  exception handling to its original exception handler.               *
 ************************************************************************/

    CALL RQ$SET$EXCEPTION$HANDLER       (@x$handler,
                                         @status);
          •
          •  }    Typical PL/M-86 Statements
          •
END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The exception$mode parameter is greater than 3.

SET$INTERRUPT

SET$INTERRUPT assigns an interrupt handler to an interrupt level and, optionally, makes the calling task the interrupt task for the level.

```
CALL RQ$SET$INTERRUPT (level, interrupt$task$flag, interrupt$handler,
                       interrupt$handler$ds, except$ptr);
```

INPUT PARAMETERS

level — A WORD containing an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

interrupt$task$flag — A BYTE indicating the interrupt task that will be invoked by the interrupt task.

- if zero, indicates that no interrupt task is to be associated with the special level and that the new interrupt handler will not call SIGNAL INTERRUPT.

CAUTION

The code for interrupt handlers that set this interrupt$task$flag to zero should not be part of a Human Interface application that is loaded into dynamic memory. If such an application is stopped (via a CTRL/c entered at a terminal), subsequent interrupts to the vector table entry set by this system call could cause unpredictable results.

- if unequal to zero, indicates that the calling task is to be the interrupt task that will be invoked by the interrupt handler being set. The priority of the calling task is adjusted by the Nucleus according to the interrupt level being serviced. Table 8-2 lists the levels and the corresponding interrupt task priorities. Be certain that priorities set in this manner do not violate the max$priority attribute of the containing job.

The value of this parameter indicates the number of outstanding SIGNAL$INTERRUPT requests that can exist. When this limit is reached, the associated interrupt level is disabled. The maximum value for this parameter is 255 decimal. Chapter 8 describes this feature in more detail.

interrupt$handler — A POINTER to the first instruction of the interrupt handler. To obtain the proper start address for interrupt handlers written in PL/M-86, place the following instruction before the call to SET$INTERRUPT:

```
interrupt$handler
            = interrupt$ptr (inter);
```

where interrupt$ptr is a PL/M-86 built-in procedure and inter is the name of your interrupt handling procedure.

interrupt$handler$ds — A WORD which specifies the interrupt handler's data segment.

- if not zero, contains the base address of the interrupt handler's data segment. See the description of ENTER$INTERRUPT in this chapter for information concerning the significance of this parameter.

  It is often desirable for an interrupt handler to pass information to the interrupt task that it calls. The following PL/M-86 statements, when included in the interrupt task's code (with the first statement listed here being the first statement in the task's code), will extract the DS register value used by the interrupt task and make it available to the interrupt handler, which in turn can access it by calling ENTER$INTERRUPT:

```
DECLARE BEGIN WORD;    /* A DUMMY VARIABLE */

DECLARE DATA$PTR POINTER;

DECLARE DATA$ADDRESS STRUCTURE (

   OFFSET WORD,

   BASE WORD) AT (@DATA$PTR); /* THIS MAKES
            ACCESSIBLE THE TWO HALVES OF THE
            POINTER DATA$PTR */

DATA$PTR = @BEGIN;  /* PUTS THE WHOLE
            ADDRESS OF THE DATA SEGMENT INTO
            DATA$PTR AND DATA$ADDRESS */

DS$BASE = DATA$ADDRESS.BASE;

CALL RQ$SET$INTERRUPT (...,DS$BASE);
```

- if zero, indicates that the interrupt handler will load its own data segment and may not invoke ENTER$INTERRUPT.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SET$INTERRUPT system call is used to inform the Nucleus that the specified interrupt handler is to service interrupts which come in at the specified level. In a call to SET$INTERRUPT, a task must indicate whether the interrupt handler will invoke an interrupt task and whether the interrupt handler has its own data segment. If the handler is to invoke an interrupt task, the call to SET$INTERRUPT also specifies the number of outstanding SIGNAL$INTERRUPT requests that the handler can make before the associated interrupt level is disabled. This number generally corresponds to the number of buffers used by the handler and interrupt task. Refer to Chapter 8 for further information.

If there is to be an interrupt task, the calling task is that interrupt task. If there is no interrupt task, SET$INTERRUPT also enables the specified level, which must be disabled at the time of the call.

EXAMPLE

```
/*************************************************************************
 *  This example illustrates how the SET$INTERRUPT system call can be    *
 *  used.                                                                 *
 *************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

   INTERRUPT_HANDLER: PROCEDURE EXTERNAL;
   END INTERRUPT_HANDLER;

   DECLARE interrupt$level$7      LITERALLY '0000 0000 0111 1000B';
                                  /* specifies master interrupt level 7 */
   DECLARE interrupt$task$flag    BYTE;
   DECLARE interrupt$handler      POINTER;
   DECLARE data$segment           WORD;
   DECLARE status                 WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

   interrupt$task$flag = 0;       /* indicates no interrupt task on level
                                     7 */
   data$segment = 0;              /* indicates that the interrupt handler
                                     will load its own data segment */
   interrupt$handler = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                  /* points to the first instruction of
                                     the interrupt handler */

        .
        .    Typical PL/M-86 Statements
        .

/*************************************************************************
 *  An interrupt level must have an interrupt handler or an interrupt    *
 *  task assigned to it.  Invoking the SET$INTERRUPT system call, the    *
 *  calling task assigns INTERRUPT_HANDLER to interrupt level 7.         *
 *************************************************************************/

   CALL RQ$SET$INTERRUPT          (interrupt$level$7,
                                  interrupt$task$flag,
                                  interrupt$handler,
                                  data$segment,
                                  @status);

        .
        .    Typical PL/M-86 Statements
        .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	One of the following is true: • The task is already an interrupt task. • The specified level already has an interrupt handler assigned to it. • The job containing the calling task or the calling task itself is in the process of being deleted.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	One of the following is true: • The level parameter is invalid or would cause the task to have a priority not allowed by its job. • The PIC corresponding to the specified level is not configured.

SET$OS$EXTENSION

The SET$OS$EXTENSION system call either enters the address of an entry (or function) procedure in the Interrupt Vector Table or it deletes such an entry.

CAUTION

This system call should not be used by Human Interface applications that are loaded into dynamic memory. If such an application is deleted (via a CTRL/c entered at a terminal), subsequent interrupts to the vector table entry set by this system call could cause unpredictable results.

```
CALL RQ$SET$OS$EXTENSION (os$extension, start$address, except$ptr);
```

INPUT PARAMETERS

os$extension	A BYTE designating the entry of the interrupt vector table to be set or reset. This value must be between 224 and 255 (decimal), inclusive. The values in the range 192 to 223 are valid, but are reserved for Intel use.
start$address	A POINTER to the first instruction of an entry (or function) procedure. If start$address contains a zero value, the specified interrupt vector table entry is being reset (deallocated).

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SET$OS$EXTENSION system call sets or resets any one of the 32 operating system extension entries in the interrupt vector table. An entry must be reset before its contents can be changed. An attempt to set an already set entry causes an E$CONTEXT exceptional condition.

EXAMPLE

```
/***************************************************************************
 *  This example illustrates how the SET$OS$EXTENSION system call can    *
 *  be used to reset an entry in the Interrupt Vector Table.  The        *
 *  example assumes that the entry for the level (number 250) was set    *
 *  earlier by another procedure.                                        *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE vector$entry$250        LITERALLY '250';
    DECLARE reset                   LITERALLY '0';
    DECLARE status                  WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

          •
          •  }   Typical PL/M-86 Statements
          •

/***************************************************************************
 *  The calling task invokes the SET$OS$EXTENSION system call to reset   *
 *  entry 250 (decimal) of the Interrupt Vector Table.                   *
 **************************************************************************/

    CALL RQ$SET$OS$EXTENSION        (vector$entry$250,
                                     reset,
                                     @status);
          •
          •  }   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	An attempt is being made to set an entry that already is set.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$PARAM	The OS$extension byte value is less than 192.

SET$POOL$MIN

SET$POOL$MIN sets a job's pool$min attribute.

```
CALL RQ$SET$POOL$MIN (new$min, except$ptr);
```

INPUT PARAMETER

new$min	A WORD indicating the pool$min attribute of the calling task's job. • if OFFFFH, indicates that the pool$min attribute of the calling task's job is to be set equal to that job's pool$max attribute. • if less than OFFFFH, contains the new value of the pool$min attribute of the calling task's job. This new value must not exceed that job's pool$max attribute.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SET$POOL$MIN system call sets the pool$min attribute of the calling task's job. The new value must not exceed that job's pool$max attribute. When the pool$min attribute is made larger than the current pool size, the pool is not enlarged until the additional memory is needed.

EXAMPLE

```
/*************************************************************************
 * This example illustrates how the SET$POOL$MIN system call can be     *
 * used.                                                                *
 *************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

   DECLARE new$min                 WORD;
   DECLARE status                  WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

   new$min = OFFFFH;               /* sets pool$min attribute of calling
                                   task's job equal to job's pool$max
                                   attribute */
        •
        •  }   Typical PL/M-86 Statements
        •

/*************************************************************************
 * In this example the pool$min attribute of the calling task's job    *
 * is to be set equal to that job's pool$max attribute.                *
 *************************************************************************/

   CALL RQ$SET$POOL$MIN            (new$min,
                                   @status);
        •
        •  }   Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	The new$min parameter is not OFFFFH, yet is greater than the pool$max attribute of the calling task's job.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.

SET$PRIORITY

The SET$PRIORITY system call changes the priority of a task.

```
CALL RQ$SET$PRIORITY (task, priority, except$ptr);
```

INPUT PARAMETERS

task	A TOKEN for the task whose priority is to be changed. A zero value specifies the invoking task.
priority	A BYTE containing the task's new priority. A zero value specifies the maximum priority of the specified task's containing job.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SET$PRIORITY system call allows the priority of a noninterrupt task to be altered dynamically.

If the priority parameter is set to the zero, the task's new priority is its containing job's maximum priority. Otherwise, the priority parameter contains the new priority of the specified task. The new priority, if explicitly specified, must not exceed its containing job's maximum priority.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the SET$PRIORITY system call can be     *
 *  used to change the priority of a task.                               *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */
    TASK_CODE: PROCEDURE EXTERNAL;
    END TASK_CODE;

    DECLARE TOKEN                 LITERALLY 'SELECTOR';
                                  /* if your PL/M compiler does not
                                     support this variable type,
                                     declare TOKEN a WORD */
    DECLARE task$token            TOKEN;
    DECLARE priority$level$66     LITERALLY '66';
    DECLARE priority$level$0      LITERALLY '0';
    DECLARE start$address         POINTER;
    DECLARE data$seg              WORD;
    DECLARE stack$pointer         POINTER;
    DECLARE stack$size$512        LITERALLY '512'; /* new task's stack
                                                      size is 512 bytes */
    DECLARE task$flags            WORD;
    DECLARE status                WORD;
    DECLARE job                   WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    start$address = @TASK_CODE;   /* pointer to first instruction of
                                     interrupt task */
    data$seg = 0;                 /* task sets up own data seg */
    stack$pointer = 0;            /* automatic stack allocation  */
    task$flags = 0;               /* designates no floating-point
                                     instructions */
          .
          .  }   Typical PL/M-86 Statements
          .

/************************************************************************
 *  In this example, the calling task creates a task whose priority is  *
 *  to be changed.  The new task initially has a priority level 66.     *
 ************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$66,
                                  start$address,
                                  data$seg,
                                  stack$pointer,
                                  stack$size$512,
                                  task$flags,
                                  @status);
```

```
/*************************************************************************
 *  The calling task in this example does not need to invoke the        *
 *  CATALOG$OBJECT system call to ensure the successful use of the      *
 *  SET$PRIORITY system call.  To allow other tasks access to the new   *
 *  task, however, requires that the task's object token be cataloged.  *
 *************************************************************************/

    CALL RQ$CATALOG$OBJECT          (job,
                                     task$token,
                                     @(12, 'TASK_CODE'),
                                     @status);

          .
          .  }   Typical PL/M-86 Statements
          .

/*************************************************************************
 *  The new task (whose code is labeled TASK_CODE) is not an interrupt  *
 *  task, so its priority may be changed dynamically by invoking the    *
 *  SET$PRIORITY system call.                                           *
 *************************************************************************/

    CALL RQ$SET$PRIORITY            (task$token,
                                     priority$level$0,
                                     @status);

          .
          .  }   Typical PL/M-86 Statements
          .

/*************************************************************************
 *  Once the need for the higher priority is no longer present, the     *
 *  priority of the new task can be changed back to its original        *
 *  priority by invoking SET$PRIORITY a second time.                    *
 *************************************************************************/

    CALL RQ$SET$PRIORITY            (task$token,
                                     priority$level$66,
                                     @status);

          .
          .  }   Typical PL/M-86 Statements
          .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	An attempt is being made to change the priority of an interrupt task.
E$EXIST	The task parameter does not refer to an existing object.

E$LIMIT	The priority parameter contains a priority value that is higher than the maximum priority of the specified task's containing job.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$TYPE	The task parameter does not contain a token for a task.

SIGNAL$EXCEPTION

The SIGNAL$EXCEPTION system call is invoked by OS extensions to signal the occurrence of an exceptional condition.

```
CALL RQ$SIGNAL$EXCEPTION(exception$code, param$num, stack$pointer,
                         reserved$param, NPX$status$word, except$ptr);
```

INPUT PARAMETERS

exception$code	A WORD containing the code (see list in Appendix B) for the exceptional condition detected.
param$num	A BYTE containing the number of the parameter which caused the exceptional condition. If no parameter is at fault, param$num equals zero.
stack$pointer	A WORD which, if not zero, must contain the value of the stack pointer saved on entry to the operating system extension (see the entry procedure in Chapter 10 for an example). The top five words in the stack (where BP is at the top of the stack) must be as follows:

FLAGS	Saved by software interrupt
CS	to OS extension
IP	
DS	Saved by OS extension
BP	on entry

Upon completion of SIGNAL$EXCEPTION, control is returned to either of two instructions. If stack$pointer contains a zero, control returns to the instruction following the call to SIGNAL$EXCEPTION. Otherwise, control returns to the instruction identified in CS and IP.

reserved$word	A WORD reserved for Intel use. Set this parameter to zero.
NPX$status$word	A WORD containing the status of the 8087 NPX.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

Operating system extensions use the SIGNAL$EXCEPTION system call to signal the occurrence of exceptional conditions. Depending on the exceptional condition and the calling task's exception mode, control may or may not pass directly to the task's exception handler.

If the exception handler does not get control, the exceptional condition code is returned to the calling task. The task can then access the code by checking the contents of the word pointed to by the except$ptr parameter <u>for its call</u> (not for the call to SIGNAL$EXCEPTION).

EXAMPLE

```
/**********************************************************************
 *  This example illustrates how the SIGNAL$EXCEPTION system call can  *
 *  be used to signal the occurrence of the exceptional condition      *
 *  E$CONTEXT.                                                         *
 **********************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

    DECLARE e$context              LITERALLY '5H';
    DECLARE param$num              BYTE;
    DECLARE stack$pointer          WORD;
    DECLARE reserved$word          LITERALLY '0';
    DECLARE status                 WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    param$num = 0;                 /* no parameter at fault */
    stack$pointer = 0;             /* return control to instruction
                                      following call */
         •
         •   }   Typical PL/M-86 Statements
         •
```

```
/**************************************************************************
 *  In this example the SIGNAL$EXCEPTION system call is invoked by       *
 *  extensions of the Operating System to signal the occurence of an     *
 *  E$CONTEXT exceptional condition.                                     *
 *************************************************************************/

    CALL RQ$SIGNAL$EXCEPTION        (e$context,
                                     param$num,
                                     stack$pointer,
                                     reserved$word,
                                     reserved$word,
                                     @status);
            .
            .  }   Typical PL/M-86 Statements
            .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CONFIGURED	This system call is not part of the present configuration.

SIGNAL$INTERRUPT

SIGNAL$INTERRUPT is used by an interrupt handler to activate an interrupt task.

```
CALL RQ$SIGNAL$INTERRUPT (level, except$ptr);
```

INPUT PARAMETER

level — A WORD containing an interrupt level which is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call. All exceptional conditions must be processed in-line, as control does not pass to an exceptional handler.

DESCRIPTION

An interrupt handler uses SIGNAL$INTERRUPT to start up its associated interrupt task. The interrupt task runs in its own environment with higher (and possibly the same) level interrupts enabled, whereas the interrupt handler runs in the environment of the interrupted task with all interrupts disabled. The interrupt task can also make use of exception handlers, whereas the interrupt handler always receives exceptions in-line.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the SIGNAL$INTERRUPT system call can    *
 *  be used to activate an interrupt task.                                *
 **************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);     /* Declares all system calls */

   DECLARE the$first$word        WORD;
   DECLARE interrupt$level$7     LITERALLY '0000 0000 0111 1000B';
                                 /* specifies master interrupt level 7 */
   DECLARE interrupt$task$flag   BYTE;
   DECLARE interrupt$handler     POINTER;
   DECLARE data$segment          WORD;
   DECLARE status                WORD;
   DECLARE interrupt$status      WORD;
   DECLARE ds$pointer            POINTER;
   DECLARE PTR$OVERLAY           LITERALLY 'STRUCTURE (offset   WORD,
                                                      base     WORD)';
                                 /* establishes a structure for
                                    overlays */
   DECLARE ds$pointer$ovly       PTR$OVERLAY AT (@ds$pointer);
                                 /* using the overlay structure, the
                                    base address of the interrupt
                                    handler's data segment is
                                    identified */

INTERRUPT_HANDLER: PROCEDURE INTERRUPT 59 PUBLIC;

          •
          •  }     Typical PL/M-86 Statements
          •

/**************************************************************************
 *  The calling interrupt handler invokes the ENTER$INTERRUPT system     *
 *  call which loads a base address value (defined by                    *
 *  ds$pointer$ovly.base) into the data segment register.  This          *
 *  register provices a mechanism for the interrupt handler to pass      *
 *  data to the interrupt task to be started up by the SIGNAL$INTERRUPT *
 *  system call.                                                         *
 **************************************************************************/

   CALL RQ$ENTER$INTERRUPT       (interrupt$level$7,
                                 @interrupt$status);
   CALL INLINE_ERROR_PROCESS     (interrupt$status);

          •
          •  }     Typical PL/M-86 Statements
          •
```

```
/************************************************************************
 *  The interrupt handler uses SIGNAL$INTERRUPT to start up its         *
 *  associated interrupt task.                                          *
 ************************************************************************/
    CALL RQ$SIGNAL$INTERRUPT      (interrupt$level$7,
                                   @interrupt$status);
    CALL INLINE_ERROR_PROCESS     (interrupt$status);

END INTERRUPT_HANDLER;

INLINE_ERROR_PROCESS: PROCEDURE;
    IF interrupt$status <> E$OK THEN
        DO;
            •
            •  }   In-line Error Processing PL/M-86 Statements
            •
        END;

END INLINE_ERROR_PROCESS;

SAMPLE_PROCEDURE:
    PROCEDURE;

    ds$pointer = @the$first$word; /* a dummy identifier used to point to
                                     interrupt handler's data segment */
    data$segment = ds$pointer$ovly.base;
                                  /* identifies the base address of the
                                     interrupt handler's data segment */
    intrpt$handlr$addrs = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                  /* points to the first instruction of
                                     the interrupt handler */
    interrupt$task$flag = 01H;    /* indicates that calling task is to be
                                     interrupt task */
          •
          •  }   Typical PL/M-86 Statements
          •

/************************************************************************
 *  By first invoking the SET$INTERRUPT system call, the calling task   *
 *  sets up an interrupt level and becomes the interrupted task for     *
 *  level 7.                                                            *
 ************************************************************************/

    CALL RQ$SET$INTERRUPT         (interrupt$level$7,
                                   interrupt$task$flag,
                                   interrupt$handler,
                                   data$segment,
                                   @status);
          •
          •  }   Typical PL/M-86 Statements
          •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	There is not an interrupt task assigned to the specified level.
E$INTERRUPT$ SATURATION	The interrupt task has accumulated the maximum allowable number of SIGNAL$INTERRUPT requests. This is an informative message only. It does not indicate an error.
E$INTERRUPT$ OVERFLOW	The interrupt task has accumulated more than the maximum allowable number of SIGNAL$INTERRUPT requests. It had reached its saturation point and then called ENABLE to allow the handler to receive further interrupt signals. It subsequently received an additional SIGNAL$INTERRUPT request before calling WAIT$INTERRUPT.
E$LIMIT	An overflow has occurred because the interrupt task has received more than 255 SIGNAL$INTERRUPT requests.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

SLEEP

SLEEP puts the calling task to sleep.

```
CALL RQ$SLEEP (time$limit, except$ptr);
```

INPUT PARAMETER

time$limit	A WORD indicating the conditions in which the calling task is to be put to sleep. • if not zero and not OFFFFH, causes the calling task to go to sleep for that many clock intervals, after which it will be awakened. The length of a clock interval is configurable. Refer to the iRMX 86 CONFIGURATION GUIDE for further information. • if zero, causes the calling task to be placed on the list of ready tasks, immediately behind all tasks of the same priority. If there are no such tasks, there is no effect and the calling task continues to run. • if OFFFFH, is invalid.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The SLEEP system call has two uses. One use places the calling task in the asleep state for a specific amount of time. The other use allows the calling task to defer to the other ready tasks with the same priority. When a task defers in this way it is placed on the list of ready tasks, immediately behind those other tasks of equal priority.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the SLEEP system call can be used.    *
 ************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);      /* Declares all system calls */

   DECLARE time$limit             WORD;
   DECLARE status                 WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

   time$limit = 100;              /* sleep for 100 clock ticks */
        •
        •  }    Typical PL/M-86 Statements
        •

/************************************************************************
 *  The calling task puts itself in the asleep state for 100 clock     *
 *  ticks by invoking the SLEEP system call.                           *
 ************************************************************************/

   CALL RQ$SLEEP                  (time$limit,
                                  @status);
        •
        •  }    Typical PL/M-86 Statements
        •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The time$limit parameter contains the invalid value OFFFFH.

SUSPEND$TASK

SUSPEND$TASK increases by one the suspension depth of a task.

```
CALL RQ$SUSPEND$TASK (task, except$ptr);
```

INPUT PARAMETER

task	A TOKEN specifying the task whose suspension depth is to be incremented. • if not zero, contains a token for the task whose suspension depth is to be incremented. • if zero, indicates that the calling task is suspending itself.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTIONS

The SUSPEND$TASK system call increases by one the suspension depth of the specified task. If the task is already in either the suspended or asleep-suspended state, its state is not changed. If the task is in the ready or running state, it enters the suspended state. If the task is in the asleep state, it enters the asleep-suspended state.

SUSPEND$TASK can not be used to suspend interrupt tasks.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the SUSPEND$TASK system call can be    *
 *  used to increase the suspension depth of a non-interrupt task.      *
 ************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    TASK_CODE: PROCEDURE EXTERNAL;
    END TASK_CODE;
    DECLARE TOKEN                LITERALLY 'SELECTOR';
                                 /* if your PL/M compiler does not
                                    support this variable type,
                                    declare TOKEN a WORD */
    DECLARE task$token           TOKEN;
    DECLARE priority$level$200   LITERALLY '200';
    DECLARE start$address        POINTER;
    DECLARE data$seg             WORD;
    DECLARE stack$pointer        POINTER;
    DECLARE stack$size$512       LITERALLY '512'; /* new task's stack
                                                     size is 512 bytes */
    DECLARE task$flags           WORD;
    DECLARE status               WORD;

SAMPLE_PROCEDURE:
    PROCEDURE;

    start$address = @TASK_CODE;  /* first instruction of the new task */
    data$seg = 0;                /* task sets up own data seg */
    stack$pointer = 0;           /* automatic stack allocation  */
    task$flags = 0;              /* designates no floating-point
                                    instructions */
          •
          • }   Typical PL/M-86 Statements
          •

/************************************************************************
 *  In order to suspend a task, a task must know the token for that     *
 *  task.  In this example, the needed token is known because the       *
 *  calling task creates the new task (whose code is labeled TASK_CODE).*
 ***********************************************************************/

    task$token = RQ$CREATE$TASK  (priority$level$200,
                                 start$address,
                                 data$seg,
                                 stack$pointer,
                                 stack$size$512,
                                 task$flags,
                                 @status);
          •
          • }   Typical PL/M-86 Statements
          •
```

```
/************************************************************************
 *  After creating the task, the calling task invokes SUSPEND$TASK.     *
 *  This system call increases by one the suspension depth of the new   *
 *  task (whose code is labeled TASK_CODE).                             *
 ************************************************************************/

    CALL RQ$SUSPEND$TASK (task$token, @status);

                        .
                        .   }   Typical PL/M-86 Statements
                        .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The task indicated by the task parameter is an interrupt task.
E$EXIST	The task parameter is not a token for an existing object.
E$LIMIT	The suspension depth for the specified task is already at the maximum of 255.
E$TYPE	The task parameter is a token for an object that is not a task.

UNCATALOG$OBJECT

UNCATALOG$OBJECT removes an entry for an object from an object directory.

```
CALL RQ$UNCATALOG$OBJECT (job, name, except$ptr);
```

INPUT PARAMETERS

job	A TOKEN indicating the job of the object directory from which an entry is to be deleted. • if not zero, the TOKEN contains a token for the job from whose object directory the specified entry is to be deleted. • if zero, the entry is to be deleted from the object directory of the calling task's job.
name	A POINTER to a STRING containing the name of the object whose entry is to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The UNCATALOG$OBJECT system call deletes an entry from the object directory of the specified job.

EXAMPLE

```
/************************************************************************
 *  This example illustrates how the UNCATALOG$OBJECT system call can   *
 *  be used.                                                             *
 ************************************************************************/

   $INCLUDE(:F1:SAMPLE.EXT);        /* Declares all system calls */

   DECLARE TOKEN                    LITERALLY 'SELECTOR';
                                    /* if your PL/M compiler does not
                                       support this variable type,
                                       declare TOKEN a WORD */
   DECLARE seg$token                TOKEN;
   DECLARE size                     WORD;
   DECLARE mbx$token                TOKEN;
   DECLARE mbx$flags                WORD;
   DECLARE no$response              LITERALLY '0';
   DECLARE status                   WORD;
   DECLARE job                      WORD;

SAMPLE_PROCEDURE:
   PROCEDURE;

   size = 64;                       /* designates new segment to contain 64
                                       bytes */
   mbx$flags = 0;                   /* designates four objects to be queued
                                       on the high performance object
                                       queue;  designates a first-in/
                                       first-out task queue. */
   job = 0;                         /* indicates objects to be cataloged
                                       into the object directory of the
                                       calling task's job */
        •
        •  }    Typical PL/M-86 Statements
        •

/************************************************************************
 *  The calling task creates a segment and a mailbox and catalogs the   *
 *  mailbox TOKEN.  The calling task then uses the TOKENs for both      *
 *  objects to send a message.                                           *
 ************************************************************************/

   seg$token = RQ$CREATE$SEGMENT  (size,
                                   @status);
   mbx$token = RQ$CREATE$MAILBOX  (mbx$flags,
                                   @status);
```

```
/************************************************************************
 *  It is not mandatory for the calling task to catalog the mailbox     *
 *  token in order to send a message.  It is necessary, however, to     *
 *  catalog the mailbox token if a task in another job is to receive    *
 *  the message.                                                        *
 ************************************************************************/

    CALL RQ$CATALOG$OBJECT           (job,
                                      mbx$token,
                                      @(3, 'MBX'),
                                      @status);

         •
         •  }   Typical PL/M-86 Statements
         •

/************************************************************************
 *  The calling task invokes the SEND$MESSAGE system call to send the   *
 *  token for the segment to the specified mailbox.                     *
 ************************************************************************/

    CALL RQ$SEND$MESSAGE             (mbx$token,
                                      seg$token,
                                      no$response,
                                      @status);
         •
         •  }   Typical PL/M-86 Statements
         •

/************************************************************************
 *  When the mailbox is no longer needed and there is no need to keep   *
 *  its token cataloged, it may be deleted by any task that knows its   *
 *  token.                                                              *
 ************************************************************************/

    CALL RQ$UNCATALOG$OBJECT         (job,
                                      @(3,'MBX'),
                                      @status):
    CALL RQ$DELETE$MAILBOX           (mbx$token,
                                      @status);

         •
         •  }   Typical PL/M-86 Statements
         •

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The specified object directory does not contain an entry with the designated name.
E$EXIST	The job parameter is neither zero nor a token for an existing object.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The first byte of the STRING pointed to by the name parameter contains a value greater than 12 or equal to 0.
E$TYPE	The job parameter is a token for an object that is not a job.

WAIT$INTERRUPT

WAIT$INTERRUPT is used by an interrupt task to signal its readiness to service an interrupt.

```
CALL RQ$WAIT$INTERRUPT (level, except$ptr);
```

INPUT PARAMETER

level — A WORD specifying an interrupt level which is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	first digit of the interrupt level (0-7)
3	if one, the level is a master level and bits 6-4 specify the entire level number if zero, the level is a slave level and bits 2-0 specify the second digit
2-0	second digit of the interrupt level (0-7), if bit 3 is zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the iRMX 86 Operating System will return the condition code generated by this system call.

DESCRIPTION

The WAIT$INTERRUPT system call is used by interrupt tasks immediately after initializing and immediately after servicing interrupts. Such a call suspends an interrupt task until the interrupt handler for the same level resumes it by invoking SIGNAL$INTERRUPT.

While the interrupt task is processing, all lower level interrupts are disabled. The associated interrupt level is either disabled or enabled, depending on the option originally specified with the SET$INTERRUPT system call. If the associated interrupt level is enabled, all SIGNAL$INTERRUPT calls that the handler makes (up to the limit specified with SET$INTERRUPT) are logged. If this count of SIGNAL$INTERRUPT calls is greater than zero when the interrupt task calls WAIT$INTERRUPT, the task is not suspended. Instead it continues processing the next SIGNAL$INTERRUPT request.

If the associated interrupt level is disabled while the interrupt task is running and the number of outstanding SIGNAL$INTERRUPT requests is less than the user-specified limit, the call to WAIT$INTERRUPT enables that level.

EXAMPLE

```
/**************************************************************************
 *  This example illustrates how the WAIT$INTERRUPT system call can be  *
 *  used to signal a task's readiness to service an interrupt.          *
 **************************************************************************/

    $INCLUDE(:F1:SAMPLE.EXT);       /* Declares all system calls */

    DECLARE TOKEN                   LITERALLY 'SELECTOR';
                                    /* if your PL/M compiler does not
                                       support this variable type,
                                       declare TOKEN a WORD */
    DECLARE task$token              TOKEN;
    DECLARE priority$level$66       LITERALLY '66';
    DECLARE start$address           POINTER;
    DECLARE data$segment            WORD;
    DECLARE stack$pointer           POINTER;
    DECLARE stack$size$512          LITERALLY '512'; /* new task's stack
                                                        size is 512 bytes */
    DECLARE task$flags              WORD;
    DECLARE interrupt$level$7       LITERALLY '0000 0000 0111 1000B';
                                    /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag     BYTE;
    DECLARE interrupt$handler       POINTER;
    DECLARE interrupt$status        WORD;
    DECLARE status                  WORD;

INTERRUPT_TASK: PROCEDURE PUBLIC;

    interrupt$task$flag = 01H;      /* indicates that calling task is to
                                       be interrupt task */
    data$segment = 0;               /* use own data segment */
    intrpt$handlr$addrs = INTERRUPT$PTR (@INTERRUPT_HANDLER);
                                    /* points to the first instruction of
                                       the interrupt handler */
```

```
/***********************************************************************
 *  The first system call in this example, SET$INTERRUPT, makes the    *
 *  calling task (INTERRUPT_TASK) the interrupt task for interrupt     *
 *  level seven.                                                       *
 ***********************************************************************/

    CALL RQ$SET$INTERRUPT          (interrupt$level$7,
                                   interrupt$task$flag,
                                   interrupt$handler,
                                   data$segment,
                                   @interrupt$status);

          •
          •  }   Typical PL/M-86 Statements
          •

/***********************************************************************
 *  The calling interrupt task invokes WAIT$INTERRUPT to suspend itself *
 *  until the interrupt handler for the same level resumes the task by  *
 *  invoking the SIGNAL$INTERRUPT system call.                          *
 ***********************************************************************/

    CALL RQ$WAIT$INTERRUPT         (interrupt$level$7,
                                   @interrupt$status);

          •
          •  }   Typical PL/M-86 Statements
          •

/***********************************************************************
 *  When the interrupt task invokes the RESET$INTERRUPT system call,    *
 *  the assignment of the current interrupt handler to interrupt level  *
 *  7 is canceled and,  because an interrupt task has also been         *
 *  assigned to the line, the interrupt task is deleted.                *
 ***********************************************************************/

    CALL RQ$RESET$INTERRUPT        (interrupt$level$7,
                                   @interrupt$status);
END INTERRUPT_TASK;

SAMPLE_PROCEDURE:
    PROCEDURE;

    start$address = @INTERRUPT_TASK;  /* 1st instruction of interrupt
                                         task */
    stack$pointer = 0;             /* automatic stack allocation  */
    task$flags = 0;                /* designates no floating-point
                                      instructions */
    data$segment = 0;              /* use own data segment */

          •
          •  }   Typical PL/M-86 Statements
          •
```

```
/*************************************************************************
 *  In this example the calling task invokes the system call            *
 *  CREATE$TASK to create a task labeled INTERRUPT_TASK.                *
*************************************************************************/

   task$token = RQ$CREATE$TASK    (priority$level$66,
                                   start$address,
                                   data$segment,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);
       .
       .  }   Typical PL/M-86 Statements
       .

END SAMPLE_PROCEDURE;
```

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The calling task is not the interrupt task for the given level.
E$NOT$CON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

SYSTEM CALLS

CHAPTER 13. CONFIGURATION OF THE NUCLEUS

The Nucleus is a configurable part of the Operating System. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, Intel provides three kinds of information:

- A list of configurable options
- Detailed information about the options
- Procedures to allow you to specify your choices

The balance of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the iRMX 86 CONFIGURATION GUIDE.

SYSTEM CALLS

The Nucleus provides system calls to support a wide variety of application software activities. By using the Interactive Configuration Utility, you can selectively eliminate system calls that are not needed by your application software or by other layers being configured.

HARDWARE

When you configure your application system, you can choose from a number of hardware features that effect the Nucleus. These features allow you to use the 80130 component (refer to the iOSP 86/88 SUPPORT PACKAGE REFERENCE MANUAL for a complete description of the 80130 component), the 8253 Programmable Interrupt Timer, the 8259A Programmable Interrupt Controller, and the 8087 Numeric Processor Extension.

• 80130 component	If the 80130 component is a part of your system, you can specify its base address location, base port address, the interval between ports, and choose whether or not you wish to use its timer and interrupt features.
• 8253 PIT	If you are using the 8253 Programmable Interval Timer to provide timing for your system, you can specify its port base, the interval between ports, clock interrupt level, clock interval and clock frequency.

• 8259 PIC	If you are using the 8259A to provide interrupt control for your system, you can specify master and slave controller bases, the interval between ports, and choose edge- or level-mode interrupts for seven or fewer slaves.
• 8087 NPX	If you have a task that needs floating-point instructions, you can specify the addition of the 8087 Numeric Processor Extension to your system.

SYSTEM CHARACTERISTICS

When you configure the Nucleus, you can specify a number of characteristics that affect your system. They include:

• Parameter Validation	A system call validates input parameters by checking for the existence of objects and by verifying that the objects are of the proper type. If your system software does not include the Basic I/O System, you may exclude parameter validation from your system.
• Minimum Transfer Size	You can choose the minimum amount of memory that the Nucleus allows to be transferred between jobs.
• Default Exception Handler	You can choose from one of four options for your system default exception handler. The choices are: • Use the system dafault exception handler which deletes offending jobs. • Use the alternative system exception handler that suspends rather than deletes. • Use the Debugger as the exception handler. • Use an exception handler that you supply.
• Exception Mode	You can choose to never pass control to the exception handler, pass control only on programmer errors, pass control only on environmental conditions, or pass control on all exceptional conditions.

APPENDIX A. iRMX™ 86 DATA TYPES

The following are the data types that are recognized by the iRMX 86 Operating System:

BYTE	- An unsigned, 8-bit, binary number.
WORD	- An unsigned, two byte, binary number.
INTEGER	- A signed, two byte, binary number that is stored in two's complement form.
POINTER	- Two words containing the base of a segment and an offset, in the reverse order.
OFFSET	- A word whose value represents the distance from the base of a segment.
BASE	- A word which identifiers a range of 64K bytes.
SELECTOR	- A 16-bit quantity that is equivalent to the base portion of a POINTER. Your PL/M compiler may not support this data type.
TOKEN	- A word or selector whose value identifies an object. A token can be declared literally a WORD or a SELECTOR depending on your needs.
STRING	- A sequence of consecutive bytes. The first byte contains the number (not to exceed 12) of bytes that follow it in the string.

APPENDIX B. iRMX™ 86 TYPE CODES

Each iRMX 86 object type is known within iRMX 86 systems by means of a numeric code. For each code, there is a mnemonic name that can be substituted for the code. Table B-1 lists the types with their codes and associated mnemonics.

Table B-1. Type Codes

OBJECT TYPE	NUMERIC CODE
Job	1
Task	2
Mailbox	3
Semaphore	4
Region	5
Segment	6
Extension	7
Composite	varies from 8000H to 0FFFFH depending on the value specified in CREATE$EXTENSION

APPENDIX C. NUCLEUS MEMORY USAGE

This appendix lists the amount of memory the Nucleus requires for object creation and memory borrowing. The Nucleus obtains this memory from the calling job's memory pool when creating the specified object or implementing the memory borrowing. The values listed in this appendix reflect Release 5 of the iRMX 86 Operating System. These values are subject to change in future releases.

The Nucleus uses the following amounts of memory when it creates objects:

object	number of 16-byte paragraphs required by the Nucleus
job	3 + object directory
object directory	1 per entry in the directory
task	5 + 6 (if the task uses the 8087 NDP) + stacksize/16 (if the Nucleus allocates the stack)
mailbox	2 + size of high performance queue/4
semaphore	2
region	2
segment	1 + segsize/16
extension	2
composite	3 + number of positions available for components/8

When a job borrows memory from its parent, the Nucleus uses three 16-byte paragraphs in addition to the amount it uses for object creation. The Nucleus obtains this memory from the parent job.

INDEX

Underscored entries are primary references.

intel®

iRMX™ 86 NUCLEUS REFERENCE MANUAL
9803122-04

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ____________

NAME ____________ DATE ____________

TITLE ____________

COMPANY NAME/DEPARTMENT ____________

ADDRESS ____________

CITY ____________ STATE (COUNTRY) ____________ ZIP CODE ____________

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

~~This document is one of a series describing Intel products. Your comments on the back of this form~~ will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Pkwy.
Hillsboro, Oregon 97123

OMO Technical Publications

intel®

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.