iRMX 86™ NUCLEUS REFERENCE MANUAL

Manual Order No.: 9803122-03

REV.	REVISION HISTORY	PRINT DATE
-01	Original Issue	4/80
-02	Adds ENTER$INTERRUPT system call, corrects various technical and typographical errors, and documents Release 2 of the iRMX 86 Operating System.	11/80
-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

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

A354/581/6K DD

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 does not contain detailed information about the features and system calls reserved for system programmers. The iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL contains this information.

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.

Manual	Number
Introduction to the iRMX 86 Operating System	9803124
iRMX 86 Installation Guide	9803125
iRMX 86 Terminal Handler Reference Manual	143324
iRMX 86 Debugger Reference Manual	143323
iRMX 86 Basic I/O System Reference Manual	9803123
iRMX 86 Extended I/O System Reference Manual	143308
iRMX 86 Loader Reference Manual	143381
iRMX 86 System Programmer's Reference Manual	142721
iRMX 86 Configuration Guide	9803126
Guide to Writing Device Drivers for the iRMX 86 I/O System	142926
iRMX 86 Programming Techniques	142982
iRMX 86 Pocket Reference	142861
iSBC 86/12A Hardware Reference Manual	9803074
ISIS-II User's Guide	9800306
PL/M-86 Programming Manual	9800466
PL/M-86 Compiler Operating Introduction for 8080/8085 Based Development Systems	9800478
The 8086 Family User's Manual	9800722

CONTENTS

FIGURES

PAGE

TABLES

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 last three object types (regions, extension objects, and composite objects) are reserved for use by system programmers, and thus are not described in this manual. Refer to the iRMX 86 SYSTEM PROGRAMMER's REFERENCE MANUAL for detailed descriptions of regions, extension objects, and composite 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 five 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 running state is a special case of the ready state. 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 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 job 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.

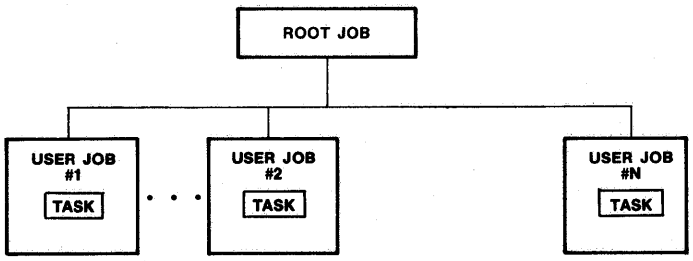

Figure 1-1. Initial Job Tree

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 two 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 the object to the mailbox, and task B must visit the mailbox, where, if an object isn't there, it has the option of waiting for any desired length of time. Sending 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 it might constitute a signal to the waiting task. Another reason to send 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.

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. Second, there are two categories of exceptional conditions: programmer errors and environmental conditions. Each task can be set up so that control goes to an exception handler in one of the following cases:

- Only when programmer errors occur
- Only when environmental conditions occur
- In both cases
- Never

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, 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 the 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

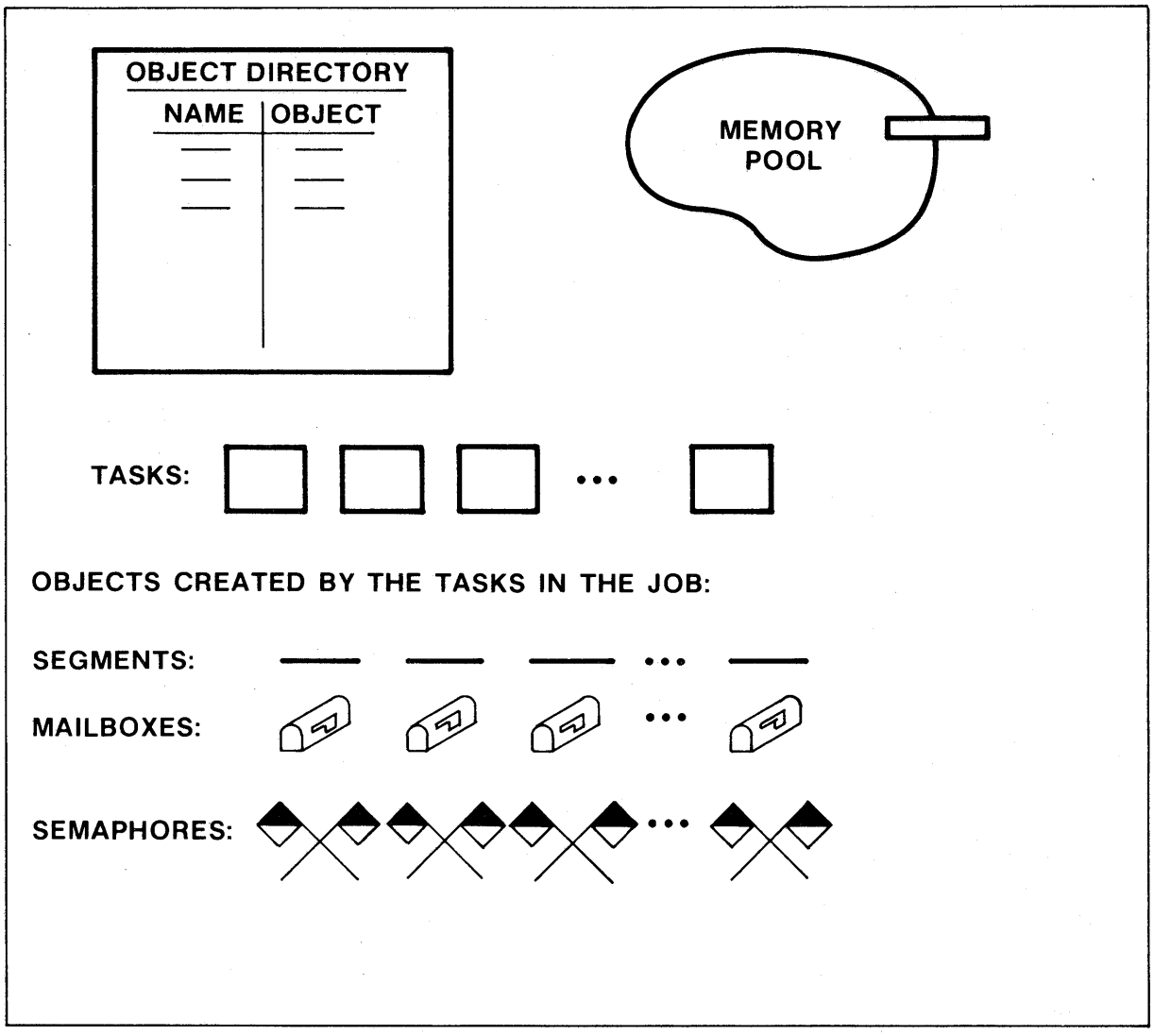

Figure 2-1. A Job

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.

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 and job B's 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 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 9.

JOB DELETION

Before a job can be deleted, all of its extension objects (see the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL) 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 CPU.

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 NDP (Numeric Data Processor), that task should not have a priority high enough to mask the interrupt level of the NDP or a deadlock situation will result. The interrupt level of the 8087 NDP 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 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.

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) The task goes from non-existence to the ready state when it is created.

(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, 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 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, 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 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 system call).

(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).

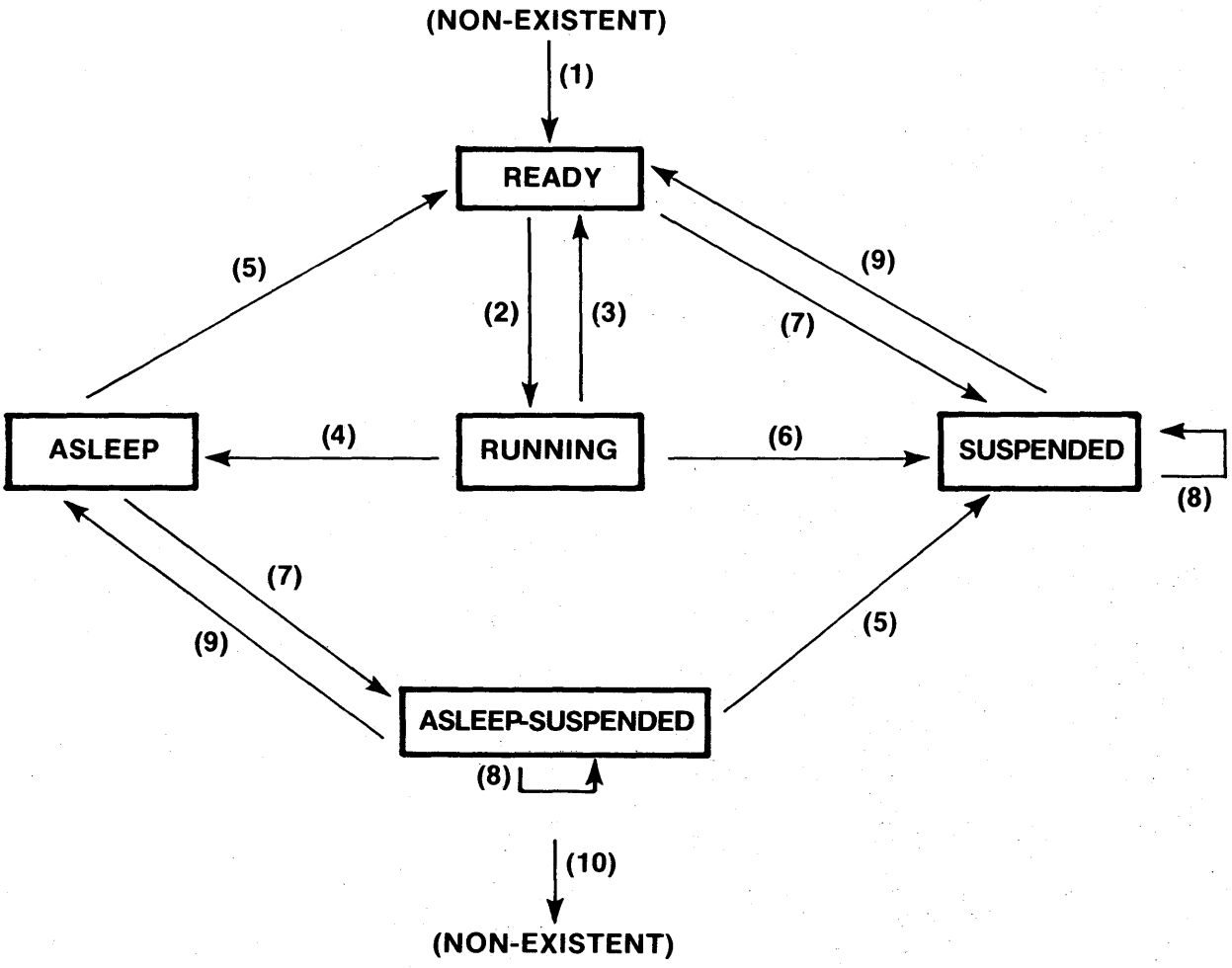

Figure 3-1. Task State Transition Diagram

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), and whether or not it is an interrupt task (see Chapter 8).

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. The task's code, however, is not a resource in this sense. It does not come from nor does it return to the task's containing 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 two kinds of exchanges: mailboxes and semaphores. If the exchange is a mailbox, one task will send an object to the mailbox; another task will go to the mailbox to receive the object. If the exchange is a semaphore, either a task is receiving units from the semaphore, or it is sending units to the semaphore.

MAILBOXES

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

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 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 9.

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. The queueing method to be used is specified for each mailbox at the time of its creation.

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, that is, memory not yet allocated in its job. 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 and the length in bytes of each segment.

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 of 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).

MEMORY POOLS

A memory pool is the amount of 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.

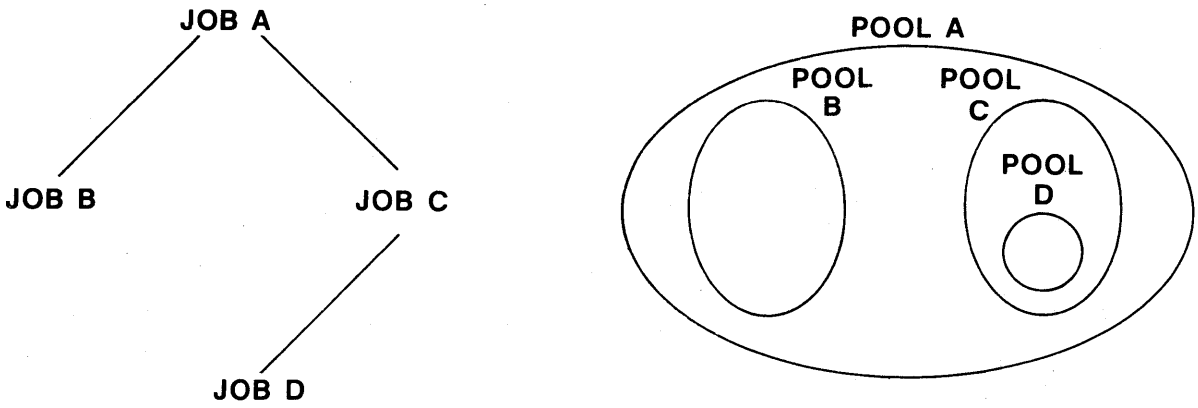

Figure 5-1. Comparison of Job and Memory Hierarchies

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, 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 allocated if it has been requested by tasks in the job or if it is on loan to a child job. Otherwise, it is unallocated.

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."

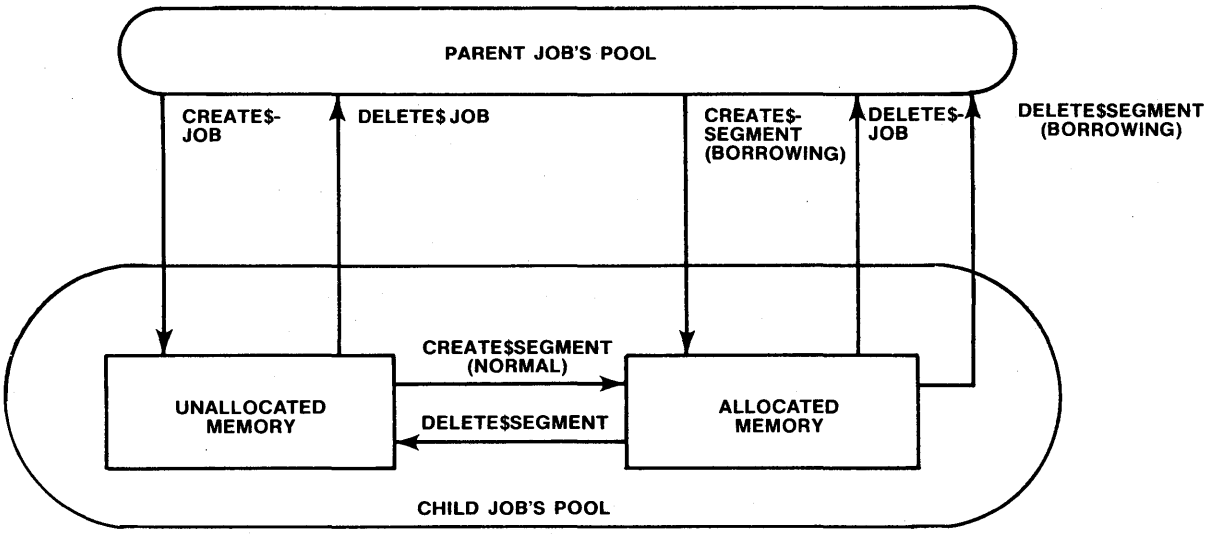

Figure 5-2. Memory Movement Diagram

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 9 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$MEM	There is not sufficient memory available to satisfy a task's request.	2H	2
E$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$STATE	A task attempted an operation which would have caused an impossible transition of a task's state.	7H	7
ENOTCONFIGURED	The system call being attempted is not not part of the present software configuration.	8H	8
E$INTERRUPT$SATURATION	An interrupt task has accumulated the maximum allowable amount of SIGNAL$INTERRUPT requests.	9H	9
E$INTERRUPT$OVERFLOW	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
EBADCALL	A task wrote over the interface library or attempted a restricted software interrupt.	8005H	32773

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 package)
2	non-maskable interrupt (used by the iSBC 957A package)
3	one byte interrupt instruction (used by the iRMX 86 Debugger and the iSBC 957A package)
4	interrupt on overflow (used by the hardware)
5	runtime array bounds error (used by compilers and assembler)
6-31	reserved
32	reserved for iRMX 86 Nucleus
33-55	reserved
56-63	reserved for external interrupts (8259A master levels)
64-127:	reserved for external interrupts (8259A slave levels)
128-183	unused (available to users)
184-190	reserved for the Nucleus
191	reserved for the iRMX 86 Debugger
192-223	reserved
224-255	described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL

INTERRUPT LEVELS

External interrupts are funneled through hardware interrupt controllers (such as the 8259A PIC). An individual 8259A 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 8259A PIC (the master) are connected to other 8259A PICs (the slaves). The eighth input line from the master 8259A 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 8259A 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 two 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.

- 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.

Regardless of the master level chosen for the system clock, the slave levels that correspond to that master level cannot be used. They do not correspond to any 8259A PIC interrupt lines. (In general, when any interrupt line of the master 8259A PIC connects directly to an interrupting device instead of to a slave PIC, the master interrupt level is used. The associated slave levels do not correspond to any interrupt lines.)

DISABLING INTERRUPTS

Occasionally you 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.

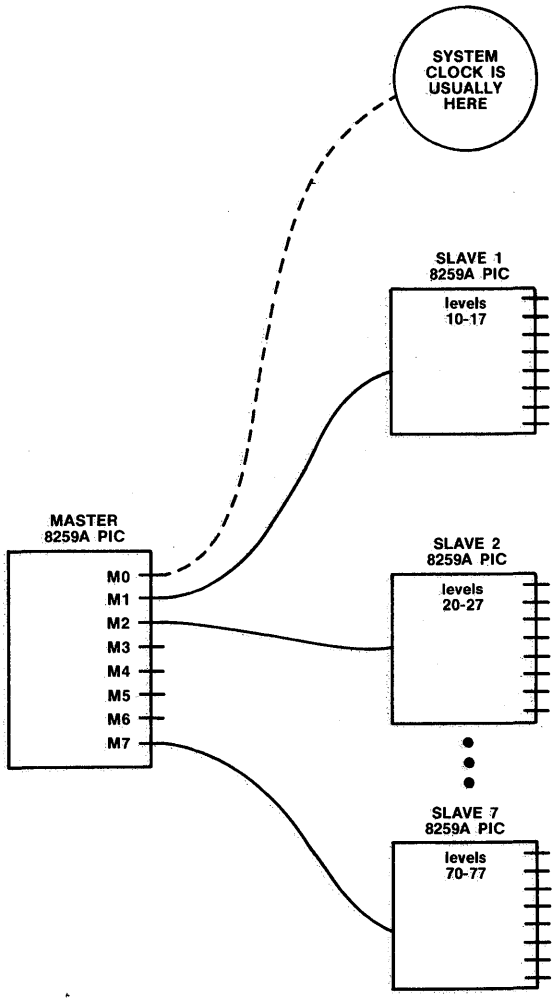

Figure 8-1. 8259A PIC Cascaded Interrupt Levels

If an interrupt signal arrives at a level that is enabled, the interrupt is recognized by the processor and control goes immediately to the interrupt handler for that level. 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 mask the level by using the DISABLE system call; later the task can unmask the level by using the ENABLE system call.
- The Nucleus disables certain interrupt levels, depending on the priority of the running task. The relationship between task priorities and disabled levels is given in Table 8-1.
- When a task makes a SET$INTERRUPT system call and designates itself as an interrupt task for a particular level, it can specify a queueing limit for unserviced interrupts. The interrupt level is disabled when the limit is reached.
- When a task makes a RESET$INTERRUPT system call to cancel the assignment of an interrupt handler to a specified level, the interrupt level is disabled.

NOTE

A task should never use the PL/M-86 DISABLE statement to disable processor interrupts. The Nucleus does not guarantee that a level so disabled will still be disabled after the task makes 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	M3 - 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	M4 - 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	M5 - M7
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	M6 - M7
83-84	51 - 77	M6 - M7
85-86	52 - 77	M6 - M7

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

Task Priority	Disabled Levels	
	Slave Levels	Master Levels
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, 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 position 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:

```
If in assembly language, save all register contents
Service the interrupt
Call EXIT$INTERRUPT
If in assembly language, restore all register contents
Return
```

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

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:

```
If in assembly language, save all register contents
Optionally, do some interrupt servicing
Call ENTER$INTERRUPT
Complete interrupt servicing
Call EXIT$INTERRUPT
If in assembly language, restore all register contents
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:

```
If in assembly language, save the register contents.
Optionally, call ENTERINTERRUPT.
Optionally, begin servicing the interrupt without system
     calls.
Call SIGNAL$INTERRUPT.
If in assembly language, restore the register contents.
Return
```

The call to SIGNAL$INTERRUPT starts up the interrupt task and enables higher (and possibly equal) priority interrupts.

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 9.

An interrupt handler executes in the environment of the interrupted task. The interrupt task, however, like any other task, has its own environment.

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

```
Call SET$INTERRUPT.
Do initialization.
Do forever;
     Call WAIT$INTERRUPT.
     Service the interrupt (system calls allowed).
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.

NOTES

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.

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

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.

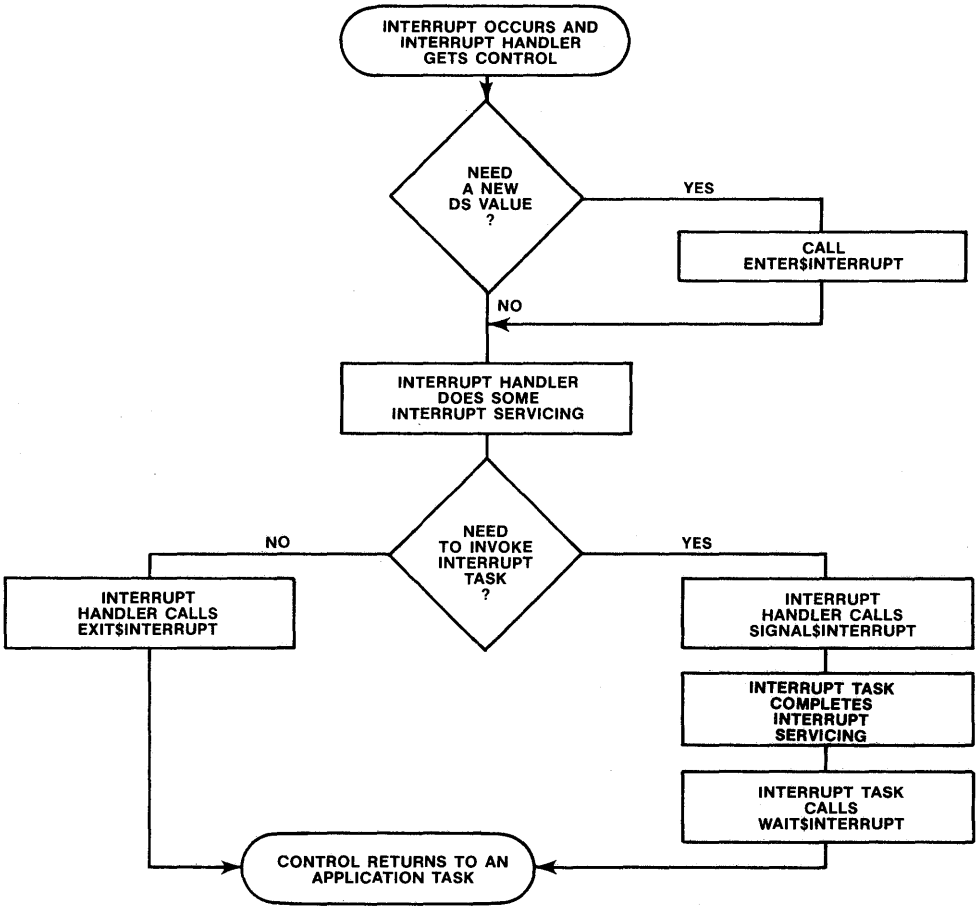

Figure 8-2. Flow Chart of Interrupt Handling

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.

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.

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. 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.

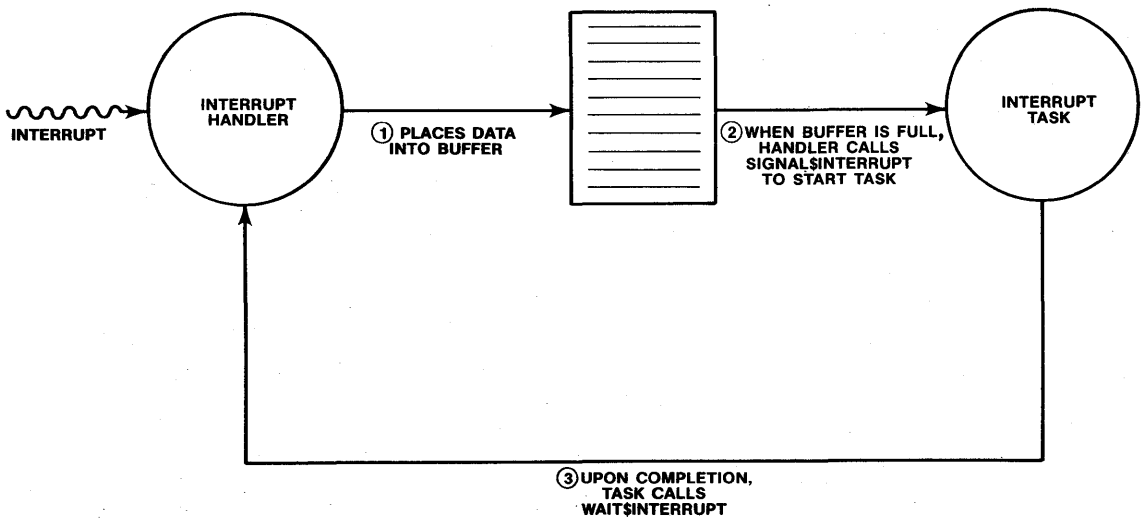

Figure 8-3. Single-Buffer Interrupt Servicing

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.

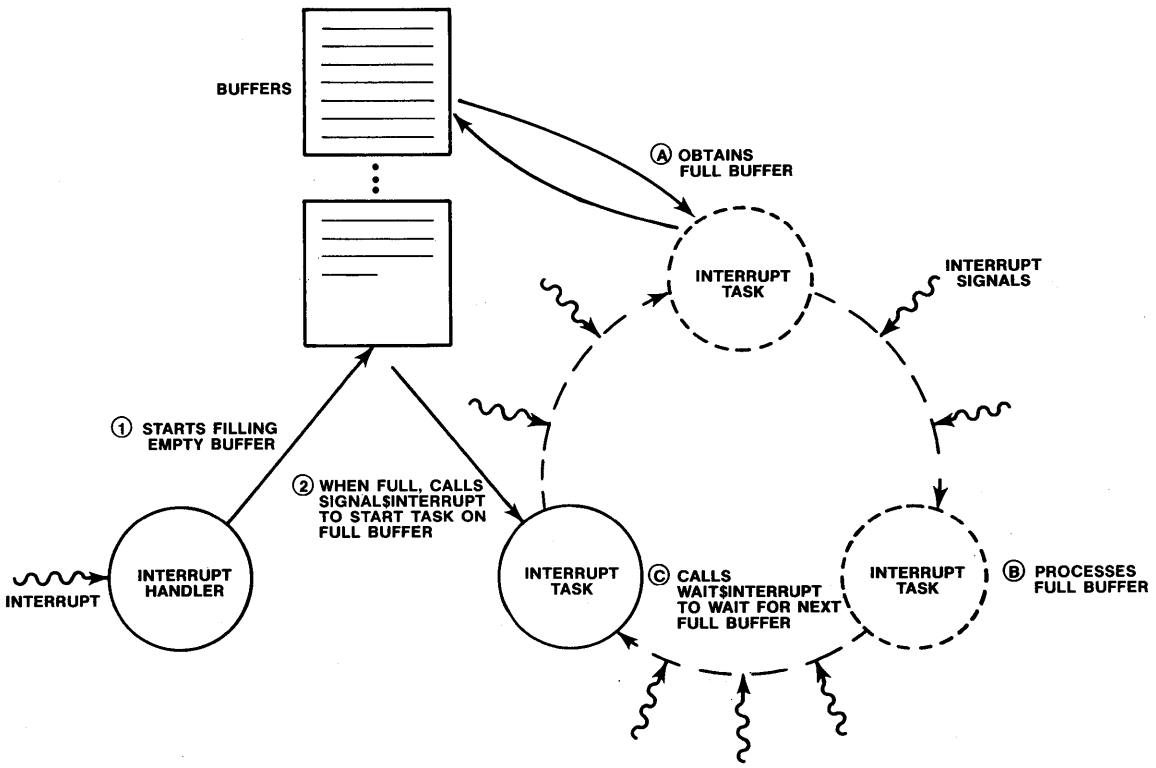

Figure 8-4. Multiple-Buffer Interrupt Servicing

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 and increments the count 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$INTERRUPT. →	Start processing first full buffer.	1
Intrpt	Process interrupt. Start filling next buffer.		
. . .			
Intrpt (A)	Process interrupt. Buffer is full. Call SIGNAL$INTERRUPT. → Count is at limit. Interrupt level is disabled.		2
		Call WAIT$INTERRUPT. Task starts processing next full buffer immediately and returns empty buffer. Interrupt level is enabled.	1
Intrpt	Process interrupt. Start filling next buffer.		
. . .			
(B)		Call WAIT$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 an 8259A PIC receives a signal from an interrupting device, it informs the processor 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	RQSETINTERRUPT (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	RQSETINTERRUPT (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$4,...);	The interrupt task begins to wait for an interrupt.	NONE
4	Level 4 device interrupts	An interrupt arrives at level M4. The interrupt handler gets control and optionally, does some servicing. The handler may service several interrupts by performing steps 4 through 6 of Figure 8-4.	M0-M7, 00-77
5	RQ$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

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	RQSETINTERRUPT (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

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

STEP	EVENTS	EXPLANATION	INTERRUPT LEVELS NECESSARILY DISABLED
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
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

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. 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 format of the call.
- Definitions of the input parameters, if any.
- Definitions of the output parameters, if any.
- A complete description of the effects of the call.
- The condition codes that can result from using the call, with a description of the possible causes of each condition.

Throughout the chapter, PL/M-86 and iRMX 86 data types, such as BYTE and STRING are used. They are always capitalized and their definitions are found in Appendix A.

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

CALLS FOR JOBS PAGE

CALLS FOR TASKS

CALLS FOR MAILBOXES

COMMAND DICTIONARY (continued)

COMMAND DICTIONARY (continued)

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 WORD which,

- 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, contains the token for the job in whose object directory the object is to be cataloged.

object — A WORD containing 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 0FFH.

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned.

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.

CATALOG$OBJECT (continued)

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.
ENOTCON-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$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 WORD which,

- 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. If the stack$ptr parameter has a base value of 0, pool$min should be at least 32 plus the value of stack$size in 16 byte paragraphs. Otherwise, pool$min should be at least 32.

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 which,

- if not 0FFFFH, contains the maximum number of objects, created by tasks in the new job, that can exist simultaneously.
- 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 which,

- if not 0FFFFH, contains the maximum number of tasks that can exist simultaneously in the new job.

CREATE$JOB (continued)

INPUT PARAMETERS
max$tasks (continued)

- if 0FFFFH, indicates that there is no limit to the number of tasks that tasks in the new job can create.

max$priority — A BYTE which,

- if not zero, 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, 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 is the default exception handler for the job. The exception$mode indicates when control is to be passed to the new task's 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.

INPUT PARAMETERS
job$flags (continued)

bit	meaning
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 which,

- if not zero, 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, 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 which,

- if not zero, contains the base address of the data segment of the new job's initial task.
- if zero, 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 which,

- if the base portion is not zero, points to the base of the user-provided stack of the new job's initial task.

CREATE$JOB (continued)

INPUT PARAMETERS
stack$ptr (continued)

- if the base portion is zero, 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 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 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 WORD containing a token for the new job.

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned.

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.

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$size is larger than 0FF0H. • The new task would exceed the object limit in the new job (that is, the max$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.

CREATE$JOB (continued)

CONDITION CODES
E$MEM (continued)

- 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 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 WORD containing a token for the new mailbox.

except$ptr — A POINTER to a WORD to which the condition code for the call is returned.

CREATE$MAILBOX (continued)

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.

CONDITION CODES

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

CREATE$SEGMENT creates a segment.

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

INPUT PARAMETER

size	A WORD which, • if not zero, 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, indicates that the size of the request is 65536 (64K) bytes.

OUTPUT PARAMETERS

segment	A WORD which contains a token for the new segment.
except$ptr	A POINTER to a WORD to which the condition code for the call is returned.

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.

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.

CREATE$SEGMENT (continued)

CONDITION CODES (continued)

ENOTCON-FIGURED	This system call is not part of the present configuration.

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 WORD containing a token for the new semaphore.

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned.

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.

CREATE$SEMAPHORE (continued)

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.
ENOTCONFIGURED	This system call is not part of the present configuration.

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 which,

- if not zero, 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, 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 which,

- if not zero, contains the base address of the new task's data segment.
- if zero, 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 which,

- if the base portion is not zero, points to the base of the new task's stack.
- if the base portion is zero, indicates that the Nucleus should allocate 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.

CREATE$TASK (continued)

INPUT PARAMETERS
stack$size (continued)

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 WORD containing a token for the new task.

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned.

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.

DESCRIPTION (continued)

- exception handler: the exception handler of the containing job.
- exception mode: the exception mode of the containing job.

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$MEM	The memory available to the calling task's job is not sufficient to create a task as specified (task descriptor, stack, and possibly 8087 area).
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The stack$size parameter is less than 16.

DELETE$JOB

DELETE$JOB deletes a job.

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

INPUT PARAMETER

job	A WORD containing a token for the job to be deleted. A value of zero specifies the calling task's job.

OUTPUT PARAMETERS

except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

DESCRIPTION

The DELETE$JOB system call deletes from the system the specified job, as well as all objects created by tasks in it. Exceptions are that jobs and extension objects (see the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL) created by tasks in the target job must be deleted prior to the call to DELETE$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.

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 the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL) which have been created by tasks in the target job.

CONDITION CODES
E$CONTEXT (continued)

- The deleting task has access to a region contained in the job to be deleted. (Refer to the iRMX SYSTEM PROGRAMMER'S REFERENCE MANUAL 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 the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL), and there is not sufficient memory for the Nucleus to send deletion messages to the appropriate deletion mailboxes.
ENOTCONFIGURED	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 WORD containing a token for the mailbox to be deleted.

OUTPUT PARAMETERS

except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

DESCRIPTION

The DELETE$MAILBOX system call deletes the specified mailbox from the system. If any tasks are queued at the mailbox at the moment of deletion, they are awakened with an E$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.

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.
ENOTCON-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$SEGMENT deletes a segment.

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

INPUT PARAMETER

segment	A WORD containing a token for the segment that is to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

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.

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.
ENOTCONFIGURED	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 WORD containing a token for the semaphore that is to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

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.

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.
ENOTCON-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 deletes a task.

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

INPUT PARAMETER

task	A WORD which, • if not zero, contains a token for the task that is to be deleted. • if zero, indicates that the calling task is to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

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.

Interrupt tasks cannot be deleted by DELETE$TASK; instead, interrupt tasks are deleted by RESET$INTERRUPT.

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.

DELETE$TASK (continued)

CONDITION CODES (continued)

ENOTCON-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 disables an interrupt level.

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

INPUT PARAMETER

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

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned. 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).

CONDITION CODES

E$OK — No exceptional conditions.

E$CONTEXT — The level indicated by the level parameter is already disabled.

DISABLE (continued)

CONDITION CODES (continued)

ENOTCON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

ENABLE enables an interrupt level.

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

INPUT PARAMETER

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

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned.

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.

CONDITION CODES

E$OK — No exceptional conditions.

ENABLE (continued)

CONDITION CODES (continued)

E$CONTEXT	At least one of the following is true: • A non-interrupt task tried to enable a level that was already enabled. • There is not an interrupt handler assigned to the specified level. • There has been an interrupt overflow on the specified level.
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

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 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

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned. 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.

One purpose of loading a new value into the DS register is to protect the contents of the interrupted task's data segment. Another purpose, if the handler is going to call an interrupt task, is that ENTER$INTERRUPT enables 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.

ENTER$INTERRUPT (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	No value had previously been specified in the call to SET$INTERRUPT.
ENOTCONFIGURED	This system call is not included in the present configuration.
E$PARAM	The level parameter is invalid.

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 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

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned. All exceptional conditions must be processed in-line. 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.

EXIT$INTERRUPT (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	No SET$INTERRUPT system call has been made for the specified level.
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

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 condition code for the call is to be returned.

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.

GET$EXCEPTION$HANDLER (continued)

CONDITION CODES

E$OK	No exceptional conditions.
ENOTCON- FIGURED	This system call is not part of the present configuration.

GET$LEVEL

GET$LEVEL returns the number of the level of the interrupt being serviced.

```
level = RQ$GET$LEVEL (except$ptr);
```

INPUT PARAMETERS

none

OUTPUT PARAMETERS

level — A WORD whose value is interpreted as follows (bit 15 is the high-order bit):

Bits	Value
15-8	undefined
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 strip away unwanted one bits, logically AND the returned value with 00FFH.

GET$LEVEL (continued)

CONDITION CODES

E$OK	No exceptional conditions.
ENOTCON- 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 condition code for the call is to be returned.

GET$POOL$ATTRIB (continued)

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.

CONDITION CODES

E$OK	No exceptional conditions.
ENOTCON-FIGURED	This system call is not part of the present configuration.

GET$PRIORITY returns the priority of a task.

```
priority = RQ$GET$PRIORITY (task, except$ptr);
```

INPUT PARAMETER

task	A WORD which, • if not zero, contains a token for the task whose priority is being requested. • if zero, indicates that 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 condition code for the call is to be returned.

DESCRIPTION

The GET$PRIORITY system call returns the priority of the specified task.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The task parameter is not a token for an existing object.
ENOTCON-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 WORD containing a token for a segment.

OUTPUT PARAMETERS

size	A WORD which, • if not zero, contains the size, in bytes, of the segment indicated by the segment parameter. • if zero, indicates that the size of the segment is 65536 (64K) bytes.
except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

DESCRIPTION

The GET$SIZE system call returns the size, in bytes, of a segment.

CONDITION CODES

E$OK	No exceptional conditons.
E$EXIST	The segment parameter is not a token for an existing object.
ENOTCON-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 returns the token requested by the calling task.

```
token = RQ$GET$TASK$TOKENS (selection, except$ptr);
```

INPUT PARAMETER

selection	A BYTE containing the request, 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 WORD containing the requested token.
except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

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.

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 WORD containing the 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 the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

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.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The object parameter is not a token for an existing object.
ENOTCONFIGURED	This system call is not part of the present configuration.

LOOKUP$OBJECT returns a token for a cataloged object.

```
object = RQ$LOOKUP$OBJECT (job, name, time$limit, except$ptr);
```

INPUT PARAMETERS

job — A WORD which,

- if not zero, contains a token for the job whose object directory is to be searched.
- if zero, indicates that 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 which,

- 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 a configuration option. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

OUTPUT PARAMETERS

object — A WORD containing the requested token.

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned.

LOOKUP$OBJECT (continued)

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.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The specified job has an object directory of size 0.
E$EXIST	One of the following is true: • The specified job was deleted while the task was waiting. • The job parameter (which is not zero) is not a token for an existing object. • The name was found, but the cataloged object has a null token (i.e. 0).
E$LIMIT	The specified object directory is full.
ENOTCON-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	Either: • 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 is a token for an object that is not a job.

OFFSPRING returns a token for each child (job) of a job.

```
token$list = RQ$OFFSPRING (job, except$ptr);
```

INPUT PARAMETER

job	A WORD containing a token for the job whose offspring are desired. A value of zero specifies the calling task's job.

OUTPUT PARAMETER

token$list	A WORD which, • if not zero, contains a token for a segment. The first word in the segment contains the number of words in the remainder of the segment. Subsequent words contain the tokens for jobs which are the immediate children of the specified job. • if zero, indicates that the specified job has no children.
except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

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.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The job parameter is not a token for an existing object.

OFFSPRING (continued)

CONDITION CODES (continued)

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.
ENOTCON-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.

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 WORD containing a token for the mailbox at which the calling task expects to receive an object token.

time$limit — A WORD which,

- if zero, indicates that the calling task is not willing to wait.
- if OFFFFH, indicates that the task will wait as long as is necessary.
- if between 0 and OFFFFH, 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 for further information.

OUTPUT PARAMETERS

object — A WORD containing the 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 has been requested by the sending task.

RECEIVE$MESSAGE (continued)

OUTPUT PARAMETERS (continued)

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 condition code for the call is to be returned.

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$MESSAGE is a token for an object that has already been deleted. To verify that the token is valid, the receiving task can call GET$TYPE. 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. If the sending task requests a response, the response$ptr parameter of RECEIVE$MESSAGE points to a token for a response exchange (either a mailbox or a semaphore). 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.

CONDITION CODES

E$OK — No exceptional conditions.

E$EXIST — Either:

- The mailbox parameter is not a token for an existing object.

CONDITION CODES
E$EXIST (continued)

- The mailbox was deleted while the task was waiting.

ENOTCONFIGURED — This system call is not part of the present configuration.

E$TIME — Either:

- 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 is 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 WORD containing 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 which, • 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

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 condition code for the call is to be returned.

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.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	Either: • The semaphore parameter is not a token for an existing object. • The semaphore was deleted while the task was waiting.
E$LIMIT	The units parameter is greater than the maximum value that had been specified for the semaphore when it was created.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$TIME	Either: • 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 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 condition code for the call is to be returned.

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).

RESET$INTERRUPT (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	There is not an interrupt handler assigned to the specified level.
ENOTCONFIGURED	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 WORD containing a token for the task whose suspension depth is to be decremented.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

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.

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$MESSAGE

SEND$MESSAGE sends an object token to a mailbox.

```
CALL RQ$SEND$MESSAGE (mailbox, object, response, except$ptr);
```

INPUT PARAMETERS

mailbox	A WORD containing a token for the mailbox to which an object token is to be sent.
object	A WORD containing an object token which is to be sent.
response	A WORD which, • 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 condition code for the call is to be returned.

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.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	One or more of the input parameters is not a token for an existing object.

SEND$MESSAGE (continued)

CONDITION CODES (continued)

E$MEM	The high performance queue is full and there is not sufficient memory in the job containing the mailbox for the Nucleus to do the housekeeping that supports a send message operation.
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$TYPE	Either: • 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 sends units to a semaphore.

```
CALL RQ$SEND$UNITS (semaphore, units, except$ptr);
```

INPUT PARAMETERS

semaphore	A WORD containing 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 condition code for the call is to be returned.

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.

CONDITION CODES

E$OK	No exceptional conditons.
E$EXIST	The semaphore parameter is not a token for an existing object.
E$LIMIT	The number of units that the calling task is trying to send would cause the semaphore's supply of units to exceed its maximum allowable supply.
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$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 condition code for the call is to be returned.

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);

DECLARE   Y   POINTER AT (@X);

DECLARE EXCEPTION WORD;

Y = @RQDEBUGGEREX;
X.MODE = ZERO$ONE$TWO$OR$THREE;
CALL  RQ$SET$EXCEPTION$HANDLER (@X, @EXCEPTION);
```

CONDITION CODES

E$OK	No exceptional conditions.
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$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 which,

- 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.
- 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.

INPUT PARAMETERS
interrupt$task$flag (continued)

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,

- 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 */
```

SET$INTERRUPT (continued)

INPUT PARAMETERS
interrupt$hander$ds (continued)

```
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 use the data segment of the interrupted task and may not call ENTER$INTERRUPT.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD to which the condition code for the call is to be returned.

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.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	One of the following is true:

- The task is already an interrupt task.

CONDITION CODES
E$CONTEXT (continued)

- 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.

ENOTCON-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$POOL$MIN

SET$POOL$MIN sets a job's pool$min attribute.

```
CALL RQ$SET$POOL$MIN (new$min, except$ptr);
```

INPUT PARAMETER

new$min	A WORD which, • if 0FFFFH, indicates that the pool$min attribute of the calling task's job is to be set equal to that job's pool$max attribute. • if less than 0FFFFH, contains the new value of the pool$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 condition code for the call is to be returned.

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.

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	The new$min parameter is not 0FFFFH, yet is greater than the pool$max attribute of the calling task's job.
ENOTCON-FIGURED	This system call is not part of the present configuration.

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 condition code for the call is to be returned. All exceptional conditions must be processed in-line. 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.

SIGNAL$INTERRUPT (continued)

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$INTERRUPT requests.
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

SLEEP puts the calling task to sleep.

```
CALL RQ$SLEEP (time$limit, except$ptr);
```

INPUT PARAMETER

time$limit — A WORD which,

- if not zero and not 0FFFFH, 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.
- if 0FFFFH, is invalid.

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD to which the condition code for the call is to be returned.

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.

CONDITION CODES

E$OK — No exceptional conditions.

SLEEP (continued)

CONDITION CODES (continued)

ENOTCON-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 WORD which, • 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 condition code for the call is to be returned.

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 cannot be used to suspend interrupt tasks.

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 WORD which, • if not zero, is a token for the job from whose object directory the specified entry is to be deleted. • if zero, indicates that 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 condition code for the call is to be returned.

DESCRIPTION

The UNCATALOG$OBJECT system call deletes an entry from the object directory of the specified job.

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.

CONDITION CODES (continued)

ENOTCON-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 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 condition code for the call is to be returned.

DESCRIPTION

The WAIT$INTERRUPT system call is used by interrupt tasks immediately after initializing and immediately after servicing interrupts. Such a call suspends an interrupt task until the interrupt handler for the same level resumes it by calling SIGNAL$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.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The calling task is not the interrupt task for the given level.
ENOTCON-FIGURED	This system call is not part of the present configuration.
E$PARAM	The level parameter is invalid.

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.

OFFSET - A word whose value represents the distance from the base of a segment.

TOKEN - A word whose value identifies an object.

POINTER- Two words containing the base of a segment and an offset, in the reverse order.

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	INTERNAL MNEMONIC	NUMERIC CODE
Job	T$JOB	1
Task	T$TASK	2
Mailbox	T$MAILBOX	3
Semaphore	T$SEMAPHORE	4
Region	T$REGION	5
Segment	T$SEGMENT	6
Extension	T$EXTENSION	7
Composite	T$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 3 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	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
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

The primary reference of each multiple-page topic is underscored.

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ 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

O.M.S. Technical Publications

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.