iRMX 86™
SYSTEM PROGRAMMER'S
REFERENCE MANUAL

Order Number: 142721-003

Copyright © 1981 Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

REV.	REVISION HISTORY	PRINT DATE
-001	Original Issue	4/81
-002	Corrects technical and typographical errors, and documents Release 2 of the iRMX 86 Operating System	9/80
-003	Adds information about the Extended I/O System, corrects technical and typographical errors, and documents Release 3 of the iRMX 86 Operating System. Debugger Information, formerly contained in this manual, is now in the iRMX 86 Debugger Reference Manual.	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

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BXP
CREDIT
i
ICE
iCS
im
Insite
Intel
Intel
Intelevision
Intellec
iRMX
iSBC
iSBX
Library Manager
MCS
Megachassis
Micromap
Multibus
Multimodule
PROMPT
Promware
RMX/80
System 2000
UPI
µScope

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

A366/581/4K DD

PREFACE

The iRMX 86 Operating System is a software package that provides real-time, multitasking capabilities for Intel iSBC 86 single board computers and any other iAPX 86- or iAPX 88-based microcomputers. This manual contains information that is separately described for system programmers. In Chapter 1, the terms system and application programmers are defined, and the reasons for making a distinction between them are explained. The remaining chapters are devoted to various kinds of information that can, if you so desire, be hidden from application programmers.

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™ Nucleus Reference Manual	9803122
iRMX 86™ Debugger Reference Manual	143323
iRMX 86™ Terminal Handler Reference Manual	143324
iRMX 86™ Basic I/O System Reference Manual	9803123
iRMX 86™ Extended I/O System Reference Manual	143308
iRMX 86™ Loader Reference Manual	143318
iRMX 86™ Human Interface Reference Manual	9803202
iRMX 86™ Configuration Guide	9803126
Guide to Writing Device Drivers for the iRMX 86™ I/O System	142926
8086/8087/8088 Macro Assembly Language Reference Manual for 8080/8085-Based Development Systems	121623
8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems	121627
8086/8087/8088 Macro Assembler Operating Instructions for 8080/8085-Based Development Systems	121624
8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems	121628

CONTENTS

PAGE

CHAPTER 1
INTRODUCTION
System and Application Programmers.................................. 1-1
About the Rest of this Manual.. 1-2

CHAPTER 2
REGIONS
Risks Involved in Sharing Data....................................... 2-1
Mutual Exclusion Using Semaphores.................................... 2-2
Mutual Exclusion Using Regions....................................... 2-3
Usefulness of Semaphores... 2-4
Regions and Deadlock... 2-4
Regions and Application Programmers.................................. 2-5
System Calls for Regions... 2-6

CHAPTER 3
OPERATING SYSTEM EXTENSIONS
Three Ways of Adding Functionality................................... 3-1
Creating an Operating System Extension............................... 3-1
  Procedures Used in Operating System Extensions..................... 3-2
    Interface Procedures... 3-6
    Entry Procedures... 3-7
    Function Procedures.. 3-10
    RQ$ERROR Procedures.. 3-10
    Linking the Procedures... 3-14
  Initializing the Interrupt Vector.................................. 3-14
Protecting Resources from Being Deleted.............................. 3-14
System Calls Used in Extending the Operating System.................. 3-15

CHAPTER 4
TYPE MANAGERS
Creating New Objects... 4-1
Manipulating Composite Objects and Extension Types................... 4-2
Deleting Composite Objects and Extension Types....................... 4-2
  Type Manager Responsibilities During DELETE$JOB.................... 4-3
  Type Manager Responsibilities during DELETE$EXTENSION.............. 4-5
  Deletion of Nested Composites...................................... 4-5
Writing a Type Manager... 4-6
Example -- A Ring Buffer Manager..................................... 4-7
  The Initialization Part.. 4-8
  The Interface Library.. 4-9
  The Entry Procedure.. 4-11
  The CREATE$RING$BUFFER Procedure................................... 4-12
  The DELETE$RING$BUFFER Procedure................................... 4-14
  The PUT$BYTE Procedure... 4-14
  The GET$BYTE Procedure... 4-15
  Epilogue... 4-16
System Calls for Type Managers....................................... 4-16

CONTENTS (continued)

PAGE

CHAPTER 5
THE I/O SYSTEM
Configuration Interface.. 5-1
  Interfacing Between Tasks and Devices.............................. 5-1
  Differences Between the Basic and Extended I/O Systems........... 5-3
  Device Connections.. 5-4
  Initialization Considerations..................................... 5-5
File Protection for Named Files..................................... 5-5
  User Objects.. 5-5
  File Access Lists... 5-7
  Access Masks for File Connections................................. 5-7
Extending a File Descriptor... 5-8

CHAPTER 6
DELETION CONSIDERATIONS... 6-1

CHAPTER 7
SYSTEM CALLS
System Call Dictionary.. 7-2
Alphabetical List of System Calls................................... 7-5
  AGETEXTENSION$DATA.. 7-5
  A$PHYSICAL$ATTACH$DEVICE.. 7-8
  A$PHYSICAL$DETACH$DEVICE.. 7-11
  ASETEXTENSION$DATA.. 7-13
  ACCEPT$CONTROL.. 7-16
  ALTER$COMPOSITE... 7-17
  CREATE$COMPOSITE.. 7-19
  CREATE$EXTENSION.. 7-21
  CREATE$REGION... 7-23
  CREATE$USER... 7-24
  DELETE$COMPOSITE.. 7-26
  DELETE$EXTENSION.. 7-27
  DELETE$REGION... 7-29
  DELETE$USER... 7-30
  DISABLE$DELETION.. 7-31
  ENABLE$DELETION... 7-33
  FORCE$DELETE.. 7-34
  INSPECT$COMPOSITE... 7-36
  INSPECT$USER.. 7-38
  LOGICAL$ATTACH$DEVICE... 7-40
  LOGICAL$DETACH$DEVICE... 7-43
  RECEIVE$CONTROL... 7-45
  SEND$CONTROL.. 7-46
  SETOSEXTENSION.. 7-47
  SET$PRIORITY.. 7-49
  SET$TIME.. 7-51
  SIGNAL$EXCEPTION.. 7-52

CONTENTS (continued)

PAGE

APPENDIX A
iRMX 86 DATA TYPES.. A-1

APPENDIX B
iRMX 86 TYPE CODES.. B-1

FIGURES

3-1. OS Extensions Without Entry Procedures.................... 3-4
3-2. OS Extension with Procedure Entry......................... 3-5
3-3. Summary of Duties of Procedures in OS Extensions........... 3-9
3-4. Handling Exceptions with an Exception Handler............. 3-11
3-5. Extension Handling Exceptions In-Line..................... 3-12
3-6. Control Flow for OS Extensions and Application Task........ 3-13
4-1. The Creation Sequence for Composite Objects................ 4-2
4-2. Type Manager Involvement in DELETE$JOB.................... 4-4
4-3. A Ring Buffer.. 4-8
5-1. Layers of Interfacing Between Tasks and a Device........... 5-2
5-2. Schematic of Software at Initialization Time............... 5-3
5-3. A System with Device and File Connections.................. 5-6
5-4. Computing the Access Mask for a File Connection............ 5-7

TABLE

3-1. Comparison of Techniques for Creating Common Functions..... 3-2

CHAPTER 1. INTRODUCTION

This chapter serves two purposes: it explains the distinction between system programmers and application programmers, and it provides a brief introduction to the contents of the rest of the manual.

SYSTEM AND APPLICATION PROGRAMMERS

For the purposes of the iRMX 86 documentation package, programmers are partitioned into two classes: application programmers and system programmers. The following paragraphs define the differences between the two classes. The distinction is somewhat artificial and may, if not useful to you, be ignored.

An <u>application programmer</u>:

- Uses a limited set of iRMX 86 capabilities and object types to achieve an applications-oriented goal.
- Is not aware of the remaining capabilities and object types.
- Does not modify the Operating System in any way.

A <u>system programmer</u>, by contrast:

- Can use all iRMX 86 capabilities and object types to achieve any desired goal.
- Can modify the operating system by creating new object types and system calls for use by system programmers and/or application programmers.

Some manuals in the iRMX 86 documentation set contain information that can safely be used by all application programmers. The others, however, including this one, present features that, if abused, could disable an application system. By documenting potentially dangerous features in separate manuals, we provide you with the opportunity of enforcing a distinction between application and system programmers.

In addition to this manual, the following manuals in the documentation package are designed for use by system programmers exclusively:

- iRMX 86 CONFIGURATION GUIDE
- GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM

ABOUT THE REST OF THIS MANUAL

The remaining chapters deal with a variety of topics.

Chapter 2 introduces regions, which are another type of exchange object.

Chapter 3 explains extending (enlarging) the Operating System.

Chapter 4 discusses a particular kind of operating system extension called a type manager. Chapter 4 also contains an example of a type manager.

Chapter 5 contains I/O System information that is deliberately not documented in either the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL or the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

Chapter 6 contains precautionary advice concerning the deletion of objects.

Chapter 7 contains the system calls that are not documented in the other reference manuals. These are the calls that could corrupt a system if used without discretion.

CHAPTER 2. REGIONS

You are probably already familiar with the concept of intertask coordination via exchanges. If you are not, you can find a general discussion in the INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM. You can also find a detailed discussion of semaphores and mailboxes in the iRMX 86 NUCLEUS REFERENCE MANUAL.

This chapter continues where the previous two discussions left off. It introduces a third type of exchange. This new iRMX 86 object type is a region, and it allows tasks to share data.

RISKS INVOLVED IN SHARING DATA

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

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

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

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

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

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

MUTUAL EXCLUSION USING SEMAPHORES

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

- Priority Bottlenecks

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

  1. Task A (low priority) obtains access to the data and continues to run.

  2. Task C (high priority) attempts to gain access, but is forced to wait at the semaphore until Task A frees the data.

  3. Task B (medium priority) awakens from a timed sleep and preempts Task A (low priority).

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

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

- Tying Up the Shared Data

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

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

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

MUTUAL EXCLUSION USING REGIONS

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

Noteworthy are the following facts regarding regions:

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

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

CAUTION

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

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

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

USEFULNESS OF SEMAPHORES

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

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

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

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

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

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

   Tasks use the ACCEPT$CONTROL system call to request immediate access. They use the RECEIVE$CONTROL system call to request access as it becomes available. Both of these system calls are described in detail in Chapter 7 of this manual.

REGIONS AND DEADLOCK

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

NOTE

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

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

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

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

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

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

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

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

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

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

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

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

REGIONS AND APPLICATION PROGRAMMERS

Knowledge of regions should not be distributed to application programmers. A careless or unscrupulous application programmer can, by abusing regions, corrupt the interaction between tasks in an application system. For instance, by creating a region and gaining access to nonexistent shared data, unscrupulous application programmers can make their tasks immune to deletion. If they never surrender access, the tasks can permanently avoid deletion.

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

SYSTEM CALLS FOR REGIONS

The following system calls manipulate regions:

- ACCEPT$CONTROL

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

- CREATE$REGION

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

- DELETE$REGION

  This system call deletes a region.

- RECEIVE$CONTROL

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

- SEND$CONTROL

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

CHAPTER 3. OPERATING SYSTEM EXTENSIONS

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

THREE WAYS OF ADDING FUNCTIONALITY

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

- Write the function as a procedure and place it in a library by using LIB86. After compiling each job that requires the function, use LINK86 to link the library to the object module for the job.

- Write the function as a task and allow application tasks to invoke the function through a mailbox-segment interface.

- Write the function as a procedure and add it to the iRMX 86 Operating System. Application programs then invoke the function by means of a system call.

The relative advantages and disadvantages of the three alternatives are summarized in Table 3-1.

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

CREATING AN OPERATING SYSTEM EXTENSION

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

TABLE 3-1. COMPARISION OF TECHNIQUES FOR CREATING COMMON FUNCTIONS

	PROCEDURE LIBRARY	TASK	OS EXTENSION
APPLICATION PROGRAMMER'S INTERFACE	SIMPLE	COMPLEX	SIMPLE
RELATIVE PERFORMANCE	GOOD (for all functions)	POOR (for quick functions) MODERATE (for slower functions)	MODERATE (for quick functions) GOOD (for slower functions)
SYNCHRONOUS or ASYNCHRONOUS CALLS	BOTH	ASYNCHRONOUS ONLY	BOTH
SYSTEM PROGRAMMER	NOT REQUIRED	NOT REQUIRED	REQUIRED
DUPLICATE CODE	Difficult to avoid	Easy to avoid	Automatically avoided
POTENTIAL FOR COSTLY MAINTENANCE	YES	NO	NO
SUPPORTS NEW OBJECT TYPES	NO	NO	YES

PROCEDURES USED IN OPERATING SYSTEM EXTENSIONS

Every OS extension is composed of at least two kinds of procedures. Figure 3-1 illustrates the simplest arrangement. The two required kinds of procedure are the following:

- Interface Procedure

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

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

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

- Function Procedure

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

Figure 3-1 depicts four OS extensions, each containing one system call. Note that the interface procedures are part of the application software and the function procedures are part of the system software. The tasks are linked to the interface procedures, but the interface procedures are not linked to the function procedures. Instead, the interface procedures pass control to the function procedures by way of the interrupt vector.

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

In Figure 3-1, the four interface procedures transfer control to the four function procedures through four separate interrupt vector entries (each of which must be numbered in the 224 to 255 range). Note that, if confined to the pattern illustrated in Figure 3-1, a system is limited to 32 customized system calls.

If a system has need for more than 32 system calls, another kind of procedure must be employed:

- Entry Procedure

  The entry procedure serves as a multiplexor for OS extensions supporting more than one system call. Figure 3-2 depicts a single OS extension with four system calls. The primary purpose of the entry procedure is to route the call from the interface procedure to the proper function procedure. Note that four interface procedures are still required to support the four system calls.

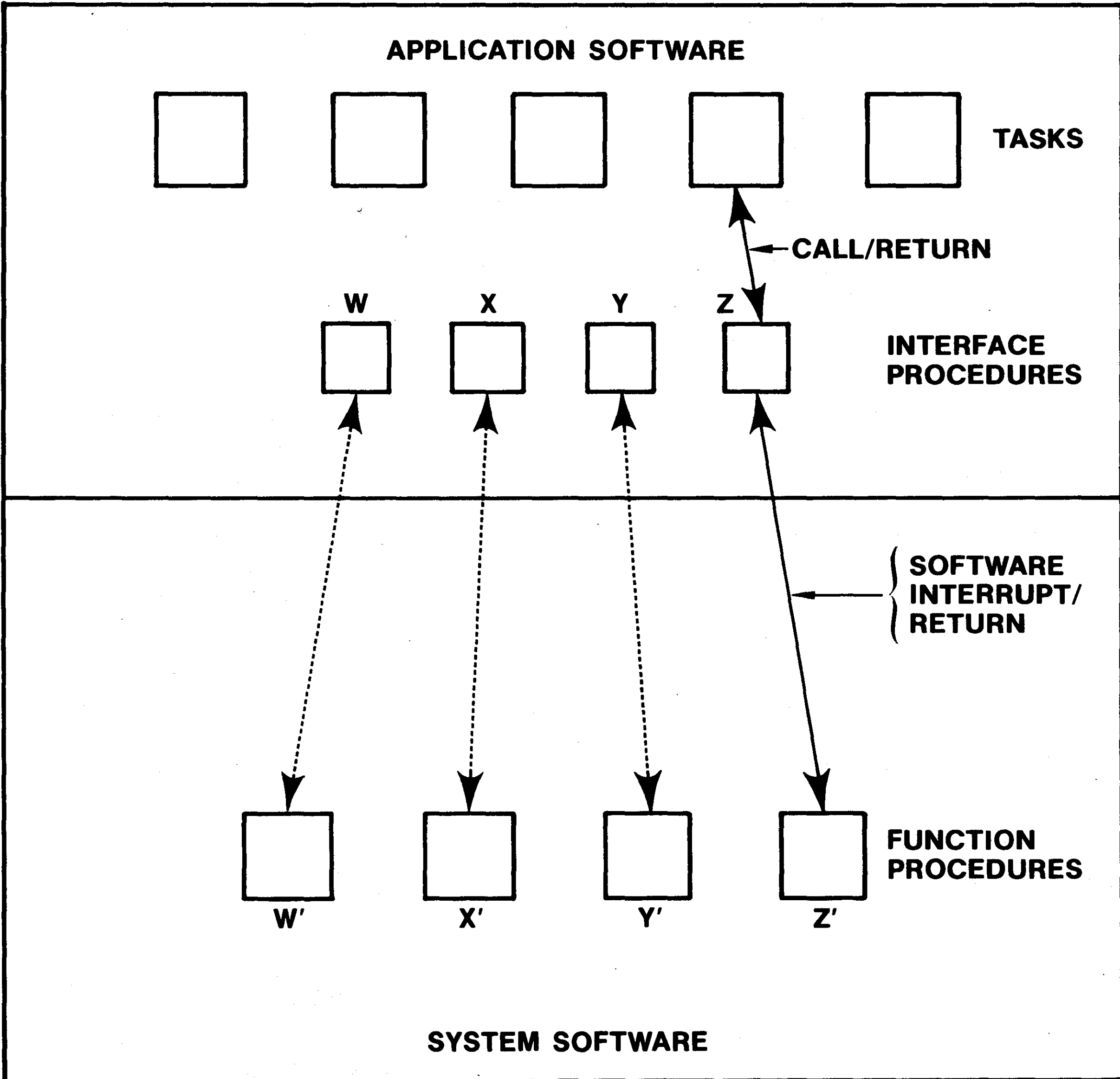


Figure 3-1. OS Extensions Without Entry Procedures

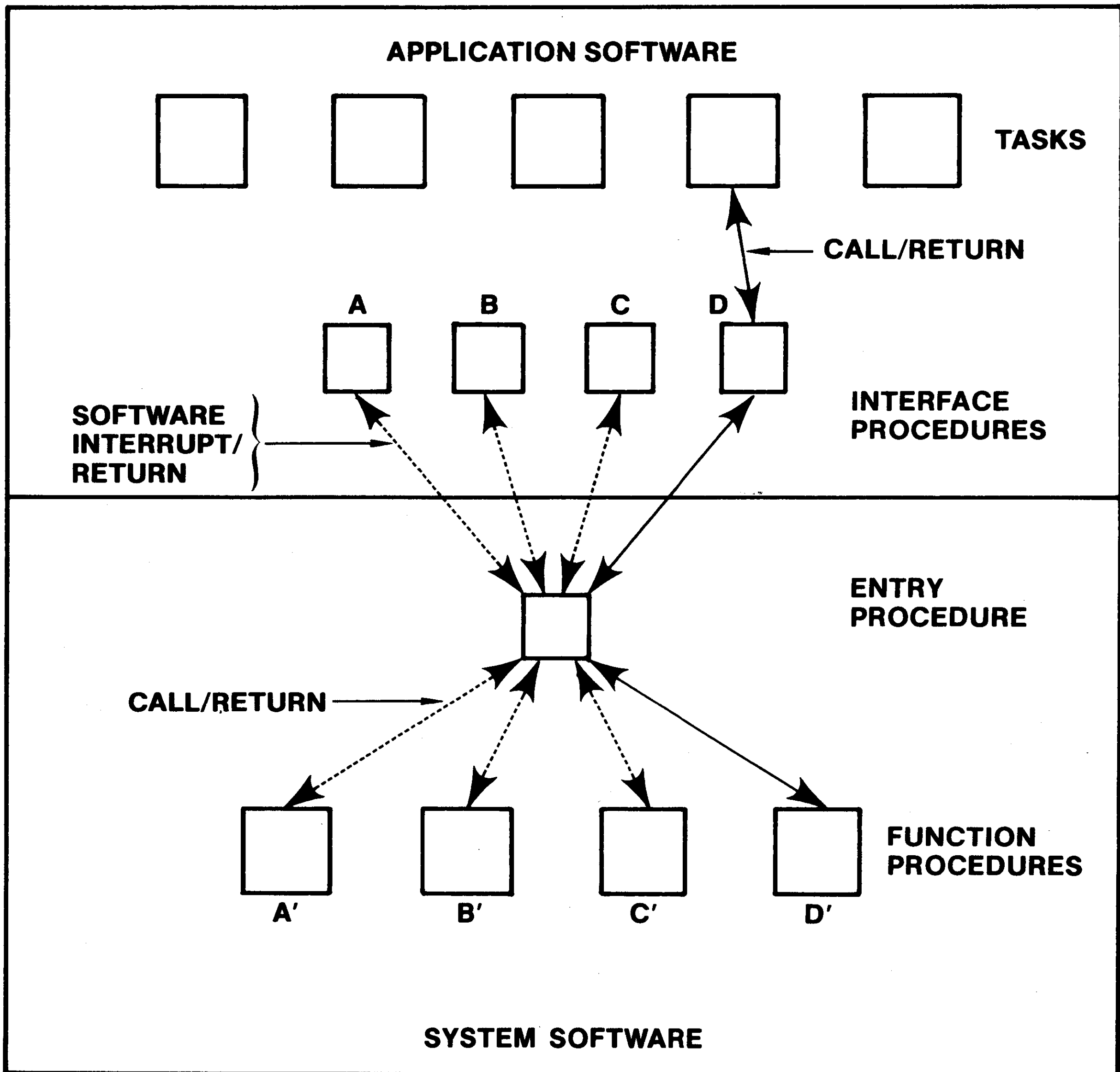


Figure 3-2. OS Extension with Procedure Entry

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

The following paragraphs describe the responsibilities of each of the kinds of procedures composing OS extensions. Figure 3-3 contains, in algorithmic form, summaries of these descriptions. Also, Chapter 4 contains an example of an OS extension that manages a customized object type.

Interface Procedures

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

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

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

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

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

If you do not want to use this default procedure, you can write your own RQ$ERROR procedure. Your RQ$ERROR procedure can perform any functions it needs in order to inform the application task of the exceptional condition. The only restriction placed on an RQ$ERROR procedure is that it should always return a value of 0FFFFH in the AX register (so that 0FFFFH is returned as a function value for your system calls that are typed procedures). An example of an alternate RQ$ERROR procedure is one that simply places 0FFFFH in AX and then does a RETURN, returning control directly to the application task to avoid the task's normal exception handler.

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

Another important purpose of interface procedures is that they compensate, on behalf of the entry or function procedures that they call, for differences between parameter-passing protocols. Three different models (COMPACT, MEDIUM, and LARGE) are available when compiling iRMX 86 tasks written in PL/M-86. Each has its own method of passing parameters. (Refer to the appropriate 8086/8087/8088 MACRO ASSEMBLER OPERATING INSTRUCTIONS manual for information regarding these methods.) By providing a library of interface procedures for each PL/M-86 model, you make the entry and function procedures independent of the PL/M-86 model in which application code is being compiled. If other languages were available, the same strategy would make the entry and function procedures independent of the language in which application code is written. The benefit of this independence is that only one entry procedure (or function procedure, if no entry procedure exists) is needed for each interrupt vector entry in your system.

Entry Procedures

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

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

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

control returns to the entry procedure from the function procedure, the entry procedure again calls SET$EXCEPTION$HANDLER to restore the original values.

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

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

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

    Place the exception code in the CX register.

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

    Return control to the interface procedure.

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

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

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

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

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

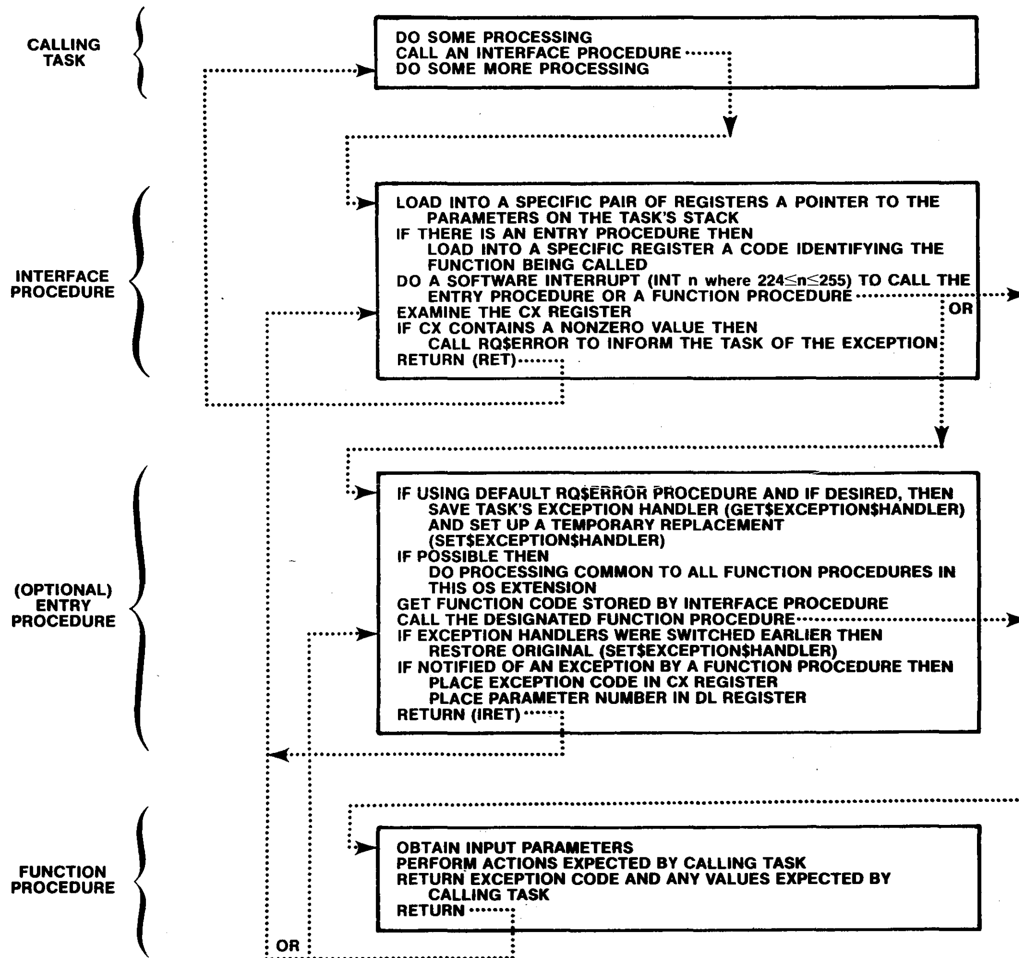


Figure 3-3. Summary of Duties of Procedures in OS Extensions

Function Procedures

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

RQ$ERROR Procedures

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

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

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

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

The Nucleus interface library, as released, contains a default RQ$ERROR procedure. The function of this RQ$ERROR procedure is to call SIGNAL$EXCEPTION to inform the calling task (or its exception handler) of the exception. This version of RQ$ERROR should be linked to application tasks to ensure that their exception handlers are called when exceptional conditions occur. Figure 3-4 illustrates the flow of control from an application task to an exception handler when the task incurs an exceptional condition.

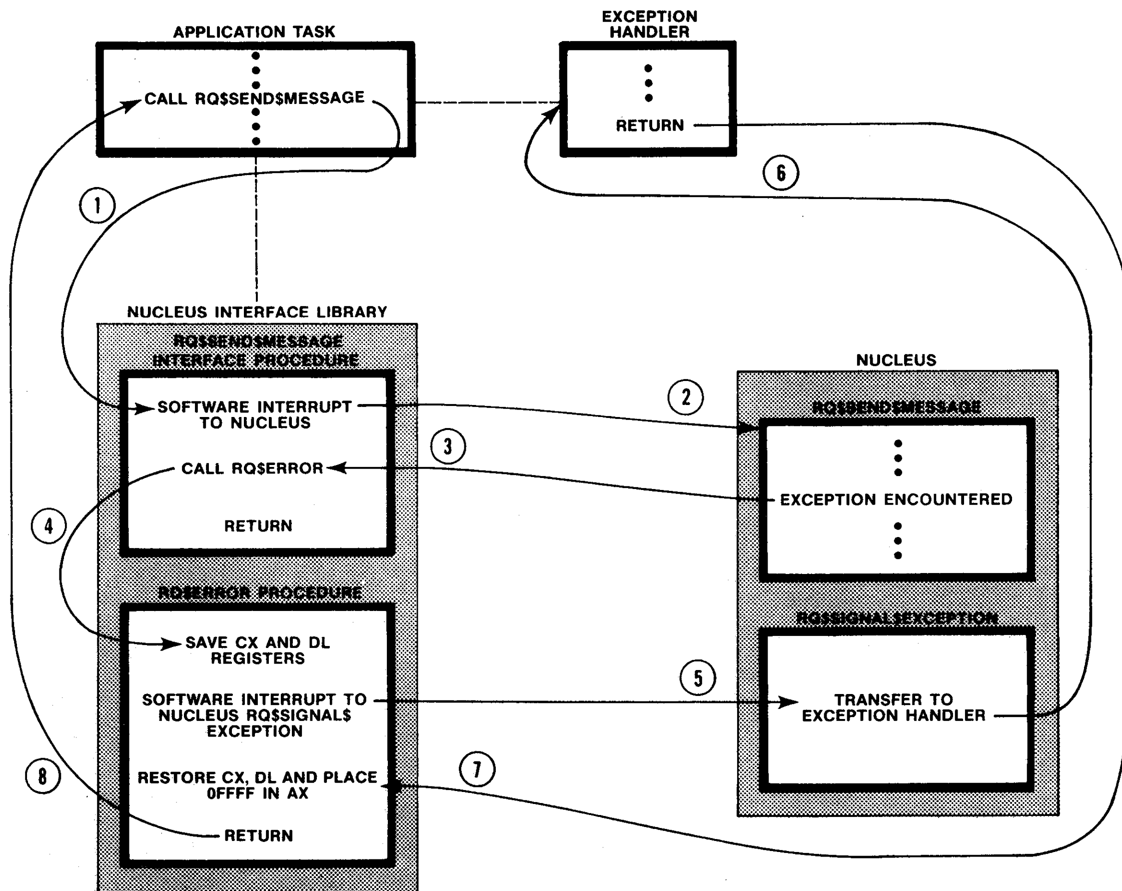


Figure 3-4. Handling Exceptions With an Exception Handler

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

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

However, if your OS extension does not have an exception handler, it should handle exceptions in-line, so that it can then return the proper exception code to the task (or OS extension) that invoked your customized system calls. You can provide this feature by linking your OS extension to a version of RQ$ERROR that does not call SIGNAL$EXCEPTION. Instead, this RQ$ERROR procedure should place 0FFFFH in the AX register (so that 0FFFFH is returned for system calls that are invoked as functions) and then do a RETURN, to return control directly to the interface library. The interface library then returns control to your OS extension, allowing the OS extension to process the exception in-line. Figure 3-5 illustrates the flow of control for an OS extension that processes its exceptions in-line. The RQ$ERROR procedure in Figure 3-5 simply sets AX and does a RETURN.

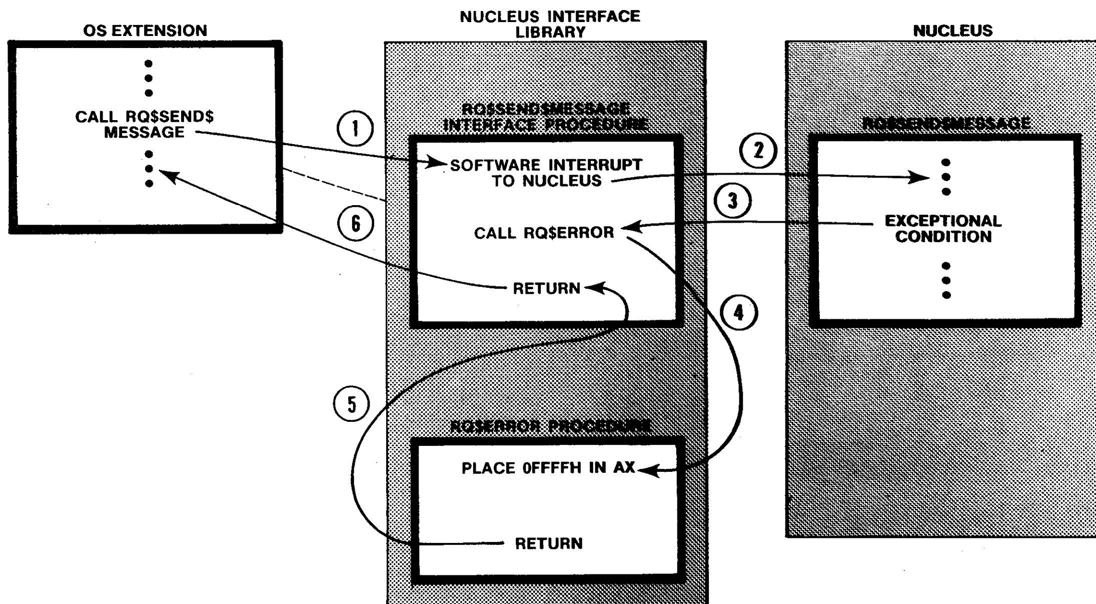


Figure 3-5. OS Extension Handling Exceptions In-Line

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

Figure 3-6 illustrates the flow of control for an OS extension that incurs an exceptional condition, processes the exception in-line, and then returns an exception to the application task that called it. Notice that both the OS extension and the application task, although not linked together, are each linked to interface libraries and an RQ$ERROR procedure. The RQ$ERROR procedure linked to the OS extension returns control back to the OS extension. The RQ$ERROR procedure linked to the application task is the default one; it calls SIGNAL$EXCEPTION.

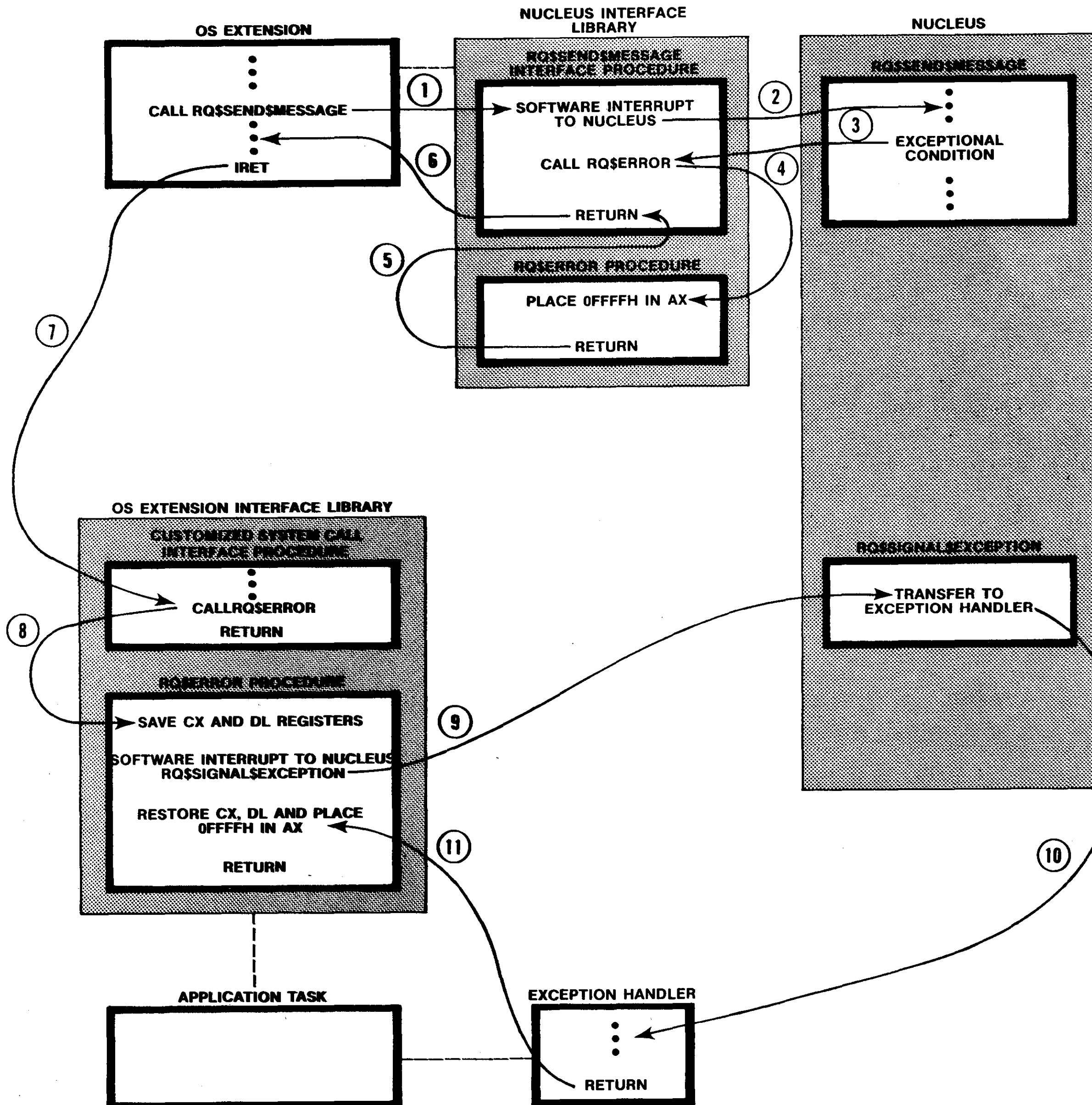


Figure 3-6. **Control Flow for OS Extension and Application Task**

Linking the Procedures

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

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

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

INITIALIZING THE INTERRUPT VECTOR

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

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

PROTECTING RESOURCES FROM BEING DELETED

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

An object can have its deletion disabled more than once. Each call to DISABLE$DELETION must be countered by a call to ENABLE$DELETION before the object can be deleted. An object's _disabling depth_ at any given moment is defined to be the number of times the object has had its

deletion disabled minus the number of times its deletion has been enabled. Usually, an object cannot be deleted until its disabling depth is zero. The lone exception is that a call to FORCE$DELETE deletes objects whose disabling depth is one. Also, calling ENABLE$DELETION for an object whose deletion depth is zero results in the E$CONTEXT exception code.

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

NOTE

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

SYSTEM CALLS USED IN EXTENDING THE OPERATING SYSTEM

The following system calls are used extensively by OS extensions:

- DISABLE$DELETION

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

- ENABLE$DELETION

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

- FORCE$DELETE

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

- SETOSEXTENSION

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

- SIGNAL$EXCEPTION

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

CHAPTER 4. TYPE MANAGERS

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

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

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

This chapter describes creating and deleting objects of new type. Chapter 3 describes extending the Operating System to include new system calls. An example appears at the end of this chapter which combines both of these operations.

CREATING NEW OBJECTS

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

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

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

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

Figure 4-1 illustrates the creation process for composite objects.

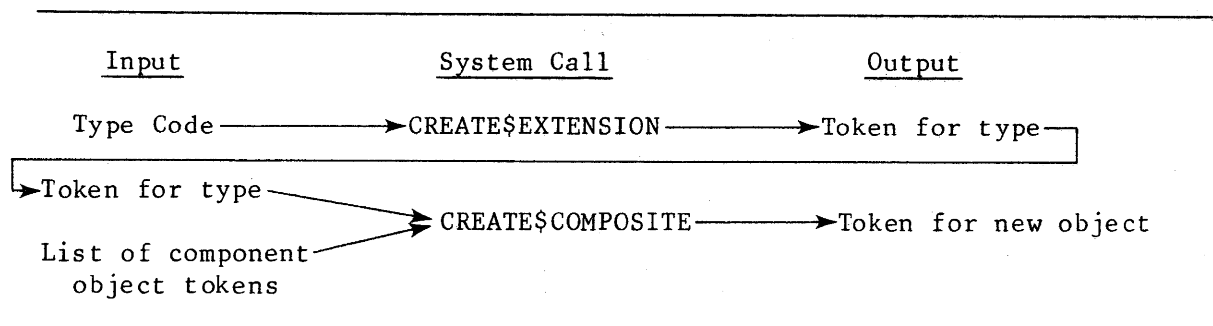


Figure 4-1. The Creation Sequence for Composite Objects

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

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

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

MANIPULATING COMPOSITE OBJECTS AND EXTENSION TYPES

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

DELETING COMPOSITE OBJECTS AND EXTENSION TYPES

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

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

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

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

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

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

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

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

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

TYPE MANAGER RESPONSIBILITIES DURING DELETE$JOB

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

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

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

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

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

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

The type manager must call DELETE$COMPOSITE each time the Nucleus sends a composite object to the deletion mailbox because DELETE$COMPOSITE serves to return control back to the Nucleus. If the type manager fails to call DELETE$COMPOSITE, the DELETE$JOB system call will not finish processing. Figure 4-2 illustrates the type manager's involvement in the DELETE$JOB process.

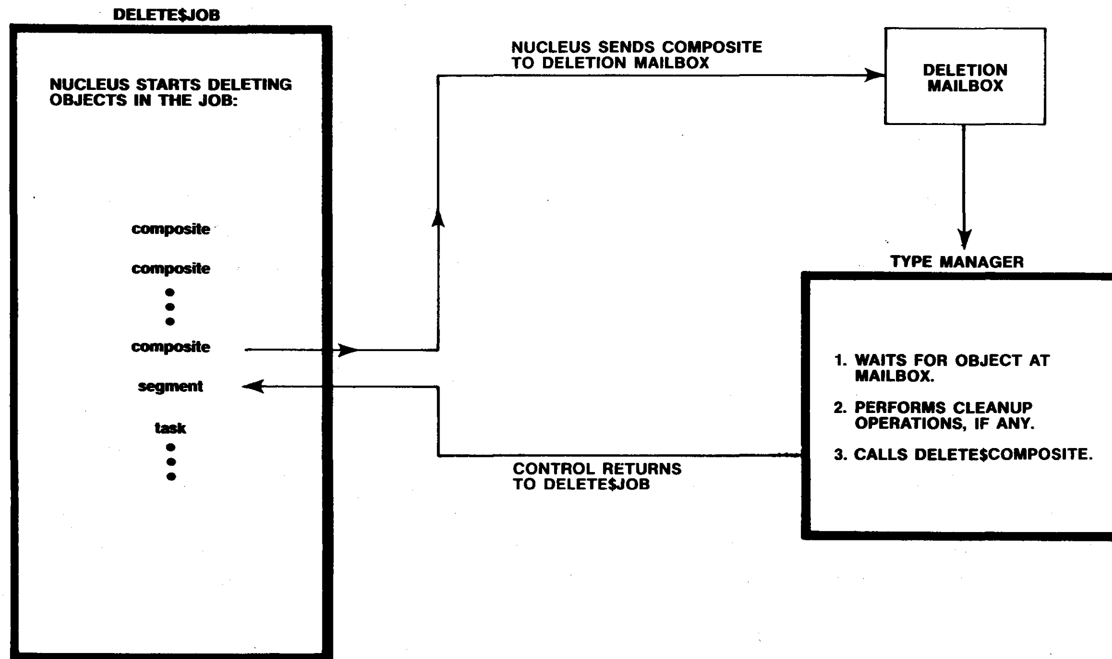


Figure 4-2. Type Manager Involvement in DELETE$JOB

Note that the type manager is not required to delete all component objects. In the course of DELETE$JOB, the Nucleus deletes any Nucleus objects in the job. The Nucleus sends any I/O System, Extended I/O System, or Human Interface objects to their respective deletion mailboxes, where the subsystems themselves delete the objects. The Nucleus sends all other composite objects to their own deletion mailboxes, where their type managers are responsible for deletion. Therefore, all the component objects are eventually deleted, provided they are in the job being deleted.

TYPE MANAGER RESPONSIBILITIES DURING DELETE$EXTENSION

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

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

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

DELETION OF NESTED COMPOSITES

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

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

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

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

WRITING A TYPE MANAGER

A type manager consists of two parts:

- The initialization part creates the type and optionally creates a deletion mailbox to which the system can send objects of the type when deleting either jobs or the type itself.

- The service part provides the system calls through which tasks can create and manipulate objects of the type.

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

The service part of the type manager is written as an operating system extension. Refer to Chapter 3 for information about operating system extensions.

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

EXAMPLE -- A RING BUFFER MANAGER

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

- The issue of exception handling is not addressed. Clearly the code supporting a system call should examine each invocation for validity, but, for brevity, the ring buffer example does not do this.

- There are no safeguards against partial creation of an object. When creating a composite object, a type manager must first create the components of the object. Occasionally, after creating some of the components, the manager might be unable to create the others. A type manager should be able to recover from this situation, usually by deleting the components already created and returning an exception code to the caller. The example, again for brevity, does not do this.

- The entry routine does not check the entry code for validity.

- The potential for problems with deletion is ignored. For this reason, you should imagine that the environment of the example is constrained in at least two ways. First, only one task will ever try to delete a ring buffer and, when it does try, no other task will be using that buffer. Second, when a job containing a task that created a ring buffer is deleted, no tasks in other jobs are using that ring buffer.

- The example has been desk-checked and the PL/M-86 portions of it have been compiled, but the example has not actually been tested.

A ring buffer is a block of memory in which bytes of data are placed at successively higher addresses. Interspersed with byte insertions are byte removals, with the restriction that the byte being removed must always be the byte that has been in the buffer for the longest time. Thus, data enters and leaves a ring buffer in a first-in-first-out manner. Ring buffers get their name from the fact that the lowest address logically follows the highest address. That is, if the last byte placed in (or retrieved from) the buffer is at its highest address, then the next byte to be placed in it (or retrieved from it) is at the lowest address. As data enters and leaves the buffer, the portion contaning data "runs" around the ring, with the pointer to the last byte out "chasing" the pointer to the last byte in. Figure 4-3 illustrates these characteristics.

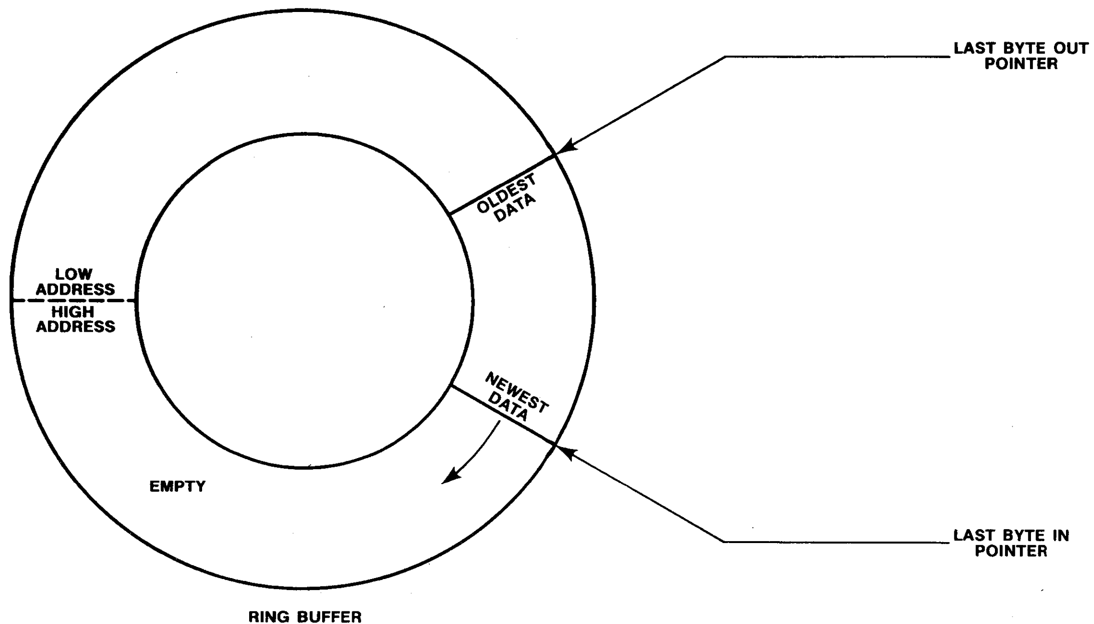


Figure 4-3. A Ring Buffer

The main (service) part of the example consists of four procedures: CREATE$RING$BUFFER, DELETE$RING$BUFFER, PUT$BYTE, and GET$BYTE. The last two procedures are for placing a character in a ring buffer, and for retrieving a character, respectively.

THE INITIALIZATION PART

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

```
    DECLARE RING$BUFFER$TYPE WORD PUBLIC;
    DECLARE DELETION$MBOX WORD PUBLIC;
    DECLARE RING$BUFFER$REGION WORD PUBLIC;
    DECLARE RING$BUFFER$MANAGER POINTER EXTERNAL;
    DECLARE RESPONSE$MBOX WORD PUBLIC;

  RING$BUFFER$INIT:  PROCEDURE;
    DECLARE DELETE$OBJECT WORD;
    DECLARE EXCEPTION WORD;
    DECLARE FIFO LITERALLY '0';
    DECLARE RB$CODE LITERALLY '8000H';
    DECLARE FOREVER LITERALLY 'WHILE 1';
    DECLARE INDEFINITELY LITERALLY '0FFFFH';

    RING$BUFFER$REGION = RQ$CREATE$REGION(FIFO, @EXCEPTION);
    DELETION$MBOX = RQ$CREATE$MAILBOX(FIFO, @EXCEPTION);
    RESPONSE$MBOX = RQ$CREATE$MAILBOX(FIFO, @EXCEPTION);
    RING$BUFFER$TYPE = RQ$CREATE$EXTENSION(RB$CODE,
                          DELETION$MBOX, @EXCEPTION);
    CALL RQ$SET$OS$EXTENSION(224, @RING$BUFFER$MANAGER,
                          @EXCEPTION);
    CALL RQ$END$INIT$TASK;
    DO FOREVER;
      DELETE$OBJECT = RQ$RECEIVE$MESSAGE(DELETION$MBOX,
                 RESPONSE$MBOX, INDEFINITELY, @EXCEPTION);
      CALL RQ$DELETE$COMPOSITE(RING$BUFFER$TYPE, DELETE$OBJECT,
                          @EXCEPTION);

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

    END RING$BUFFER$INIT;
```

The variable RING$BUFFER$MANAGER is a pointer to the entry procedure of the operating system extension.

THE INTERFACE LIBRARY

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

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


            ASSUME   CS:CGROUP
DELETERB    PROC     FAR
            PUBLIC   RQDELETERB
            PUSH     BP
            MOV      BP,SP
            LEA      SI,SS: BP+6
            MOV      BX,1             ;Code for DELETE$RING$BUFFER
            INT      224
            POP      BP
            RET      2                ;Passing one argument
DELETERB    ENDP


            ASSUME   CS:CGROUP
GETRBBYTE   PROC     FAR
            PUBLIC   RQGETBYTE
            PUSH     BP
            MOV      BP,SP
            LEA      SI,SS: BP+6
            MOV      BX,2             ;Code for GET$BYTE
            INT      224
            POP      BP
            RET      2                ;Passing one argument
GETRBBYTE   ENDP
            ASSUME   CS:CGROUP
PUTRBBYTE   PROC     FAR
            PUBLIC   RQPUTBYTE
            PUSH     BP
            MOV      BP,SP
            LEA      SI,SS: BP+6
            MOV      BX,3             ;Code for PUT$BYTE
            INT      224
            POP      BP
            RET      4                ;Passing two arguments
PUTRBBYTE   ENDP
```

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

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


DELETERB:  PROCEDURE(RING$BUFFER$TOKEN) EXTERNAL;
  DECLARE RING$BUFFER$TOKEN WORD;
END DELETERB;


GETRBBYTE:  PROCEDURE(RING$BUFFER$TOKEN) BYTE EXTERNAL;
  DECLARE RING$BUFFER$TOKEN WORD;
END GETRBBYTE;


PUTRBBYTE:  PROCEDURE(CHAR, RING$BUFFER$TOKEN) EXTERNAL;
  DECLARE CHAR BYTE;
  DECLARE RING$BUFFER$TOKEN WORD;
END PUTRBBYTE;
```

THE ENTRY PROCEDURE

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

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

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

THE CREATE$RING$BUFFER PROCEDURE

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

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

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

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

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

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

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

The CREATE$RING$BUFFER routine creates the components of the composite ring buffer object, initializes the appropriate fields, and then creates the composite object, as follows:

```
    DECLARE RING$BUFFER$TYPE WORD EXTERNAL;

  CREATE$RING$BUFFER:  PROCEDURE (PARAM$PTR) WORD PUBLIC
                                                  REENTRANT;
    DECLARE PARAM$PTR POINTER;
    DECLARE SIZE BASED PARAM$PTR WORD;
    DECLARE ASTR STRUCTURE(
              NUM$SLOTS        WORD,
              NUM$COMPONENTS   WORD,
              SEG              WORD,
              EMPTY$CT         WORD,
              FULL$CT          WORD);
    DECLARE SEG$PTR POINTER;
    DECLARE PTR$STRUC STRUCTURE(
              OFFSET           WORD,
              BASE             WORD) AT (@SEG$PTR);
    DECLARE SEGMENT BASED SEG$PTR STRUCTURE(
              SIZE             WORD,
              HEAD             WORD,
              TAIL             WORD,
              BUFFER(1)        BYTE);
    DECLARE EXCEPTION WORD;
    DECLARE RING$BUFFER WORD;
    DECLARE PRIORITY LITERALLY '1';

    ASTR.NUM$SLOTS = 3;
    ASTR.NUM$COMPONENTS = 3;
    ASTR.SEG = RQ$CREATE$SEGMENT(SIZE+6, @EXCEPTION);
    ASTR.EMPTY$CT = RQ$CREATE$SEMAPHORE(SIZE, SIZE, PRIORITY,
                                        @EXCEPTION);
    ASTR.FULL$CT = RQ$CREATE$SEMAPHORE(0, SIZE, PRIORITY,
                                        @EXCEPTION);
    PTR$STRUC.BASE = ASTR.SEG;
    PTR$STRUC.OFFSET = 0;
    SEGMENT.SIZE = SIZE;
    SEGMENT.HEAD = -1;
    SEGMENT.TAIL = 0;
    RING$BUFFER = RQ$CREATE$COMPOSITE(RING$BUFFER$TYPE, @ASTR,
                                                  @EXCEPTION);
    RETURN RING$BUFFER;
    END CREATE$RING$BUFFER;
```

The SEGMENT.HEAD variable is set to -1 because the PUT$BYTE procedure (shown later) advances this pointer before placing a character in the buffer.

THE DELETE$RING$BUFFER PROCEDURE

DELETE$RING$BUFFER can be called by any task wanting to delete a ring buffer:

```
    DECLARE RING$BUFFER$TYPE WORD EXTERNAL;

  DELETE$RING$BUFFER:  PROCEDURE(PARAM$PTR) REENTRANT PUBLIC;
    DECLARE PARAM$PTR POINTER;
    DECLARE RING$BUFFER$TOKEN BASED PARAM$PTR WORD;
    DECLARE ASTR STRUCTURE(
              NUM$SLOTS      WORD,
              NUM$COMPONENTS WORD,
              SEG            WORD,
              EMPTY$CT       WORD,
              FULL$CT        WORD);
    DECLARE EXCEPTION WORD;

    ASTR.NUM$SLOTS = 3;
    CALL RQ$INSPECT$COMPOSITE(RING$BUFFER$TYPE,
                      RING$BUFFER$TOKEN, @ASTR, @EXCEPTION);
    CALL RQ$DELETE$COMPOSITE(RING$BUFFER$TYPE,
                           RING$BUFFER$TOKEN, @EXCEPTION);
    CALL RQ$DELETE$SEGMENT(ASTR.SEG, @EXCEPTION);
    CALL RQ$DELETE$SEMAPHORE(ASTR.EMPTY$CT, @EXCEPTION);
    CALL RQ$DELETE$SEMAPHORE(ASTR.FULL$CT, @EXCEPTION);
    END DELETE$RING$BUFFER;
```

THE PUT$BYTE PROCEDURE

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

```
    DECLARE RING$BUFFER$TYPE WORD EXTERNAL;
    DECLARE RING$BUFFER$REGION WORD EXTERNAL;

  PUT$BYTE: PROCEDURE(PARAM$PTR) REENTRANT PUBLIC;
    DECLARE PARAM$PTR POINTER;
    DECLARE PARAMS BASED PARAM$PTR STRUCTURE(
              RING$BUFFER$TOKEN  WORD,
              CHAR               BYTE);
    DECLARE SIZE WORD;
    DECLARE ASTR STRUCTURE(
              NUM$SLOTS       WORD,
              NUM$COMPONENTS  WORD,
              SEG             WORD,
              EMPTY$CT        WORD,
              FULL$CT         WORD);
    DECLARE SEG$PTR POINTER;
```

```
    DECLARE PTR$STRUC STRUCTURE(
              OFFSET          WORD,
              BASE            WORD) AT (@SEG$PTR);
    DECLARE SEGMENT BASED SEG$PTR STRUCTURE(
              SIZE            WORD,
              HEAD            WORD,
              TAIL            WORD,
              BUFFER(1)       BYTE);
    DECLARE EXCEPTION WORD;
    DECLARE INDEFINITELY LITERALLY '0FFFFH';
    DECLARE UNITS$LEFT WORD;

    ASTR.NUM$SLOTS = 3;
    CALL RQ$INSPECT$COMPOSITE(RING$BUFFER$TYPE,
              PARAMS.RING$BUFFER$TOKEN, @ASTR, @EXCEPTION);
    UNITS$LEFT = RQ$RECEIVE$UNITS(ASTR.EMPTY$CT, 1,
                            INDEFINITELY, @EXCEPTION);
    CALL RQ$RECEIVE$CONTROL(RING$BUFFER$REGION,
                                      @EXCEPTION);
    PTR$STRUC.BASE = ASTR.SEG;
    PTR$STRUC.OFFSET = 0;
    SEGMENT.HEAD = ((SEGMENT.HEAD + 1) MOD SEGMENT.SIZE);
    SEGMENT.BUFFER(SEGMENT.HEAD) = PARAMS.CHAR;
    CALL RQ$SEND$CONTROL(@EXCEPTION);
    CALL RQ$SEND$UNITS(ASTR.FULL$CT, 1, @EXCEPTION);
    END PUT$BYTE;
```

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

Note also that the order of the parameters RING$BUFFER$TOKEN and CHAR is the opposite of the order of those parameters in the earlier external declaration of PUTRBBYTE. This is typical of procedures with multiple arguments and results from the use of the stack when passing parameters.

THE GET$BYTE PROCEDURE

GET$BYTE removes the oldest byte in the buffer and then advances the SEGMENT.TAIL "pointer":

```
    DECLARE RING$BUFFER$TYPE WORD EXTERNAL;
    DECLARE RING$BUFFER$REGION WORD EXTERNAL;

  GET$BYTE: PROCEDURE(PARAM$PTR) BYTE PUBLIC REENTRANT;
    DECLARE PARAM$PTR POINTER;
    DECLARE RING$BUFFER$TOKEN BASED PARAM$PTR WORD;
    DECLARE SIZE WORD;
    DECLARE ASTR STRUCTURE(
              NUM$SLOTS       WORD,
              NUM$COMPONENTS WORD,
              SEG             WORD,
```

```
            EMPTY$CT        WORD,
            FULL$CT         WORD);
    DECLARE SEG$PTR POINTER;
    DECLARE PTR$STRUC STRUCTURE(
            OFFSET          WORD,
            BASE            WORD) AT (@SEG$PTR);
    DECLARE SEGMENT BASED SEG$PTR STRUCTURE(
            SIZE            WORD,
            HEAD            WORD,
            TAIL            WORD,
            BUFFER(1)       BYTE);
    DECLARE EXCEPTION WORD;
    DECLARE CHAR BYTE;
    DECLARE INDEFINITELY LITERALLY 'OFFFFH';
    DECLARE UNITS$LEFT WORD;

    ASTR.NUM$SLOTS = 3;
    CALL RQ$INSPECT$COMPOSITE(RING$BUFFER$TYPE,
                         RING$BUFFER$TOKEN, @ASTR, @EXCEPTION);
    UNITS$LEFT = RQ$RECEIVE$UNITS(ASTR.FULL$CT, 1, INDEFINITELY,
                                                     @EXCEPTION);
    CALL RQ$RECEIVE$CONTROL(RING$BUFFER$REGION, @EXCEPTION);
    PTR$STRUC.BASE = ASTR.SEG;
    PTR$STRUC.OFFSET = 0;
    CHAR = SEGMENT.BUFFER(SEGMENT.TAIL);
    SEGMENT.TAIL = ((SEGMENT.TAIL + 1) MOD SEGMENT.SIZE);
    CALL RQ$SEND$CONTROL(@EXCEPTION);
    CALL RQ$SEND$UNITS(ASTR.EMPTY$CT, 1, @EXCEPTION);
    RETURN CHAR;
    END GET$BYTE;
```

EPILOGUE

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

SYSTEM CALLS FOR TYPE MANAGERS

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

- ALTER$COMPOSITE

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

- CREATE$COMPOSITE

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

- CREATE$EXTENSION

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

- DELETE$COMPOSITE

  This system call deletes a composite object.

- DELETE$EXTENSION

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

- INSPECT$COMPOSITE

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

CHAPTER 5. THE I/O SYSTEM

This chapter contains information enabling system programmers to provide application programmers with the facilities they need to make full use of the Basic and Extended I/O Systems. The chapter comprises the following topics:

- The configuration interface, which binds (and unbinds) file drivers to individual device units.
- The creation and deletion of user objects.
- Adding to, or obtaining information from, file descriptors.

CONFIGURATION INTERFACE

Before a task can create a connection to a file on a device, a connection must have been created to the device itself. The Basic I/O System configuration interface consists of two system calls that create and delete connections to devices. They are:

A$PHYSICAL$ATTACH$DEVICE
A$PHYSICAL$DETACH$DEVICE

A$PHYSICAL$ATTACH$DEVICE creates connections to devices for the Basic I/O System. A$PHYSICAL$DETACH$DEVICE deletes these connections.

The Extended I/O System configuration interface also consists of two system calls. They are:

LOGICAL$ATTACH$DEVICE
LOGICAL$DETACH$DEVICE

LOGICAL$ATTACH$DEVICE assigns logical names to devices and causes the device connections to be created the first time tasks try to access the devices using the logical names. LOGICAL$DETACH$DEVICE deletes the logical names and causes the device connections to be deleted when no tasks have connections to files on the device.

INTERFACING BETWEEN TASKS AND DEVICES

Figure 5-1 shows the layers of software and hardware between a device and the application tasks using files on the device.

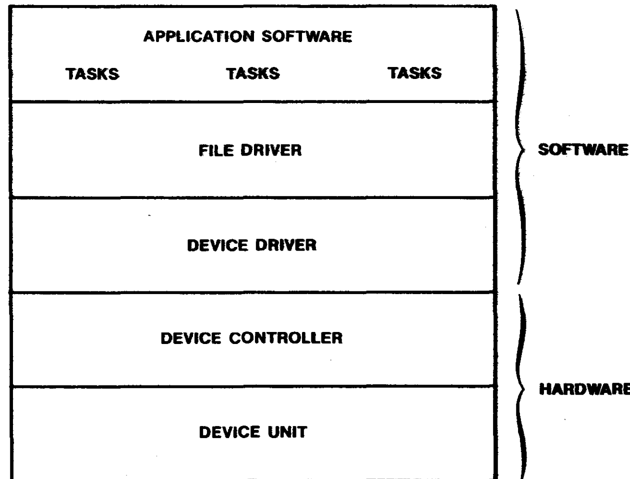


Figure 5-1. Layers of Interfacing Between Tasks and a Device

The layers shown in Figure 5-1 must be bound together. The device controller is physically bound to each of its device-units. The device driver is bound to the device controller by information residing in a Device Unit Information Block for the device. (For more on this, see the iRMX 86 CONFIGURATION GUIDE or the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM.) The application software is bound to the file drivers during the linking process. When an application system starts up, these three forms of binding are in place.

The configuration interface dynamically binds the appropriate file driver (physical, named, or stream) to the device, its controller, and its device driver. By creating this final bond dynamically, you can break it later and replace it with a bond to a different file driver. Figure 5-2 shows schematically the situation that exists when the system starts up.

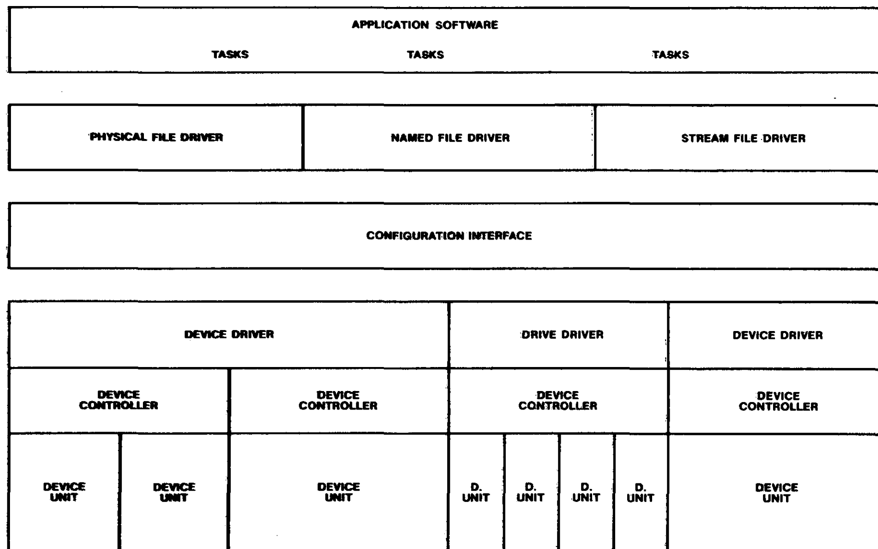


Figure 5-2. Schematic of Software at Initialization Time

DIFFERENCES BETWEEN THE BASIC AND EXTENDED I/O SYSTEMS

There are two main differences between creating and deleting device connections with the Basic I/O System calls and creating and deleting them with the Extended I/O System calls. These differences involve synchronous versus asynchronous operation and logical names.

The Basic I/O System calls, A$PHYSICAL$ATTACH$DEVICE and A$PHYSICAL$DETACH$DEVICE, are asynchronous calls which do not use logical names. When calling A$PHYSICAL$ATTACH$DEVICE to create a device connection, your task specifies the device name, the file driver, and a mailbox token as parameters. Then, due to the asynchronous nature of the call, the task must wait at the mailbox for the Basic I/O System to send it a token for the device connection. When it receives this token, the task can use it as a prefix in other Basic I/O System calls that create or attach files on the device. Later, when the task wants to delete the device connection, it can specify this token as a parameter in A$PHYSICAL$DETACH$DEVICE.

The Extended I/O System calls, LOGICAL$ATTACH$DEVICE and LOGICAL$DETACH$DEVICE, are synchronous system calls that make use of logical names. When calling LOGICAL$ATTACH$DEVICE to attach a device, your task specifies the device name and the file driver as it does with A$PHYSICAL$ATTACH$DEVICE, but it also specifies a logical name. The Extended I/O System creates a Logical Device Object and catalogs it in the root job's object directory under the logical name your task specified in the call. Your task does not have to wait at a mailbox to receive the result of the call; the call is performed synchronously. After calling LOGICAL$ATTACH$DEVICE, your tasks can use the logical name as the prefix portion of a path name in other Extended I/O System calls

that create or attach files on the device. During the first such call, the Extended I/O System creates a device connection. Later, when your task wants to delete the device connection, it can specify the logical name as a parameter in LOGICAL$DETACH$DEVICE.

There is a restriction you should be aware of when deciding whether to use A$PHYSICAL$ATTACH$DEVICE or LOGICAL$ATTACH$DEVICE to attach a device. If you use the Extended I/O System (LOGICAL$ATTACH$DEVICE) to attach a device, you must also use Extended I/O System calls to perform any functions that require you to specify a path name. These calls include:

S$ATTACH$FILE
S$CHANGE$ACCESS
S$CREATE$DIRECTORY
S$CREATE$FILE
S$DELETE$FILE
SGETFILE$STATUS
S$RENAME$FILE

You must not use the corresponding Basic I/O System calls to perform these functions. If you obey this restriction, you gain the ability to replace diskettes in a drive attached with LOGICAL$ATTACH$DEVICE without destroying the device connection. Otherwise, the device connection will be lost when the device goes off-line.

DEVICE CONNECTIONS

This section is based in large part on the following analogy: A device connection is like an electrical conduit (pipe) and the file connections to that device unit are like wires in that conduit. Figure 5-3 depicts the system of Figure 5-2 after several device connections have been created.

This figure is quite detailed and shows most of the situations that can occur. The following observations can be made:

- The device connections extend from the application software to the individual device-units.

- There is only one device connection to each connected device. Multiple tasks can share the same device connection.

- The configuration interface, which is depicted as a pile of conduits, is off to the side.

- All but one of the device units are connected. The unconnected device unit is still separated from the application software by the configuration interface.

- Different device units with the same controller can be connected via different file drivers.

- Tasks can share access to the same device through the physical file driver, and they can share access to files on the same device through the named file driver.
- There is only one device connection through the stream file driver, reflecting the fact that a single, logical device contains all stream files.

INITIALIZATION CONSIDERATIONS

A device unit must be bound to a file driver before any application tasks can successfully create file connections involving that device unit. One way of ensuring this is to see that all initial device connections are created by an initialization task or tasks. Then, so the returned tokens will be available to application tasks, the connections should be cataloged, probably in the root object directory.

FILE PROTECTION FOR NAMED FILES

The I/O System controls the access of tasks trying to use named files. For each named file, the I/O System uses the file's access list and a user object to produce an access mask it associates with a connection object for the file. The mask enables the I/O System to monitor the access rights of tasks wanting to use the file via that connection object. This file protection mechanism is not available for physical or stream files.

USER OBJECTS

A user object is an iRMX 86 object type that includes identifying information about a conceptual user (job or human) and the groups to which the user belongs. Each user object contains an array of 16-bit values called ID's. The first ID in the array is called the owner ID of the user object. User objects are created and deleted by calls to CREATE$USER and DELETE$USER, respectively. The ID's in a user object can be read by means of a call to INSPECT$USER.

In systems using the I/O System, each job can have a default user object. Tasks in the job can specify this default user object in certain system calls simply by passing a zero value as a user object parameter. In the case of jobs created by CREATEIOJOB, the default user object can be set when the job is created. SET$DEFAULT$USER can be used either to change an existing default user object or, in the case of jobs having no default user object, to establish one.

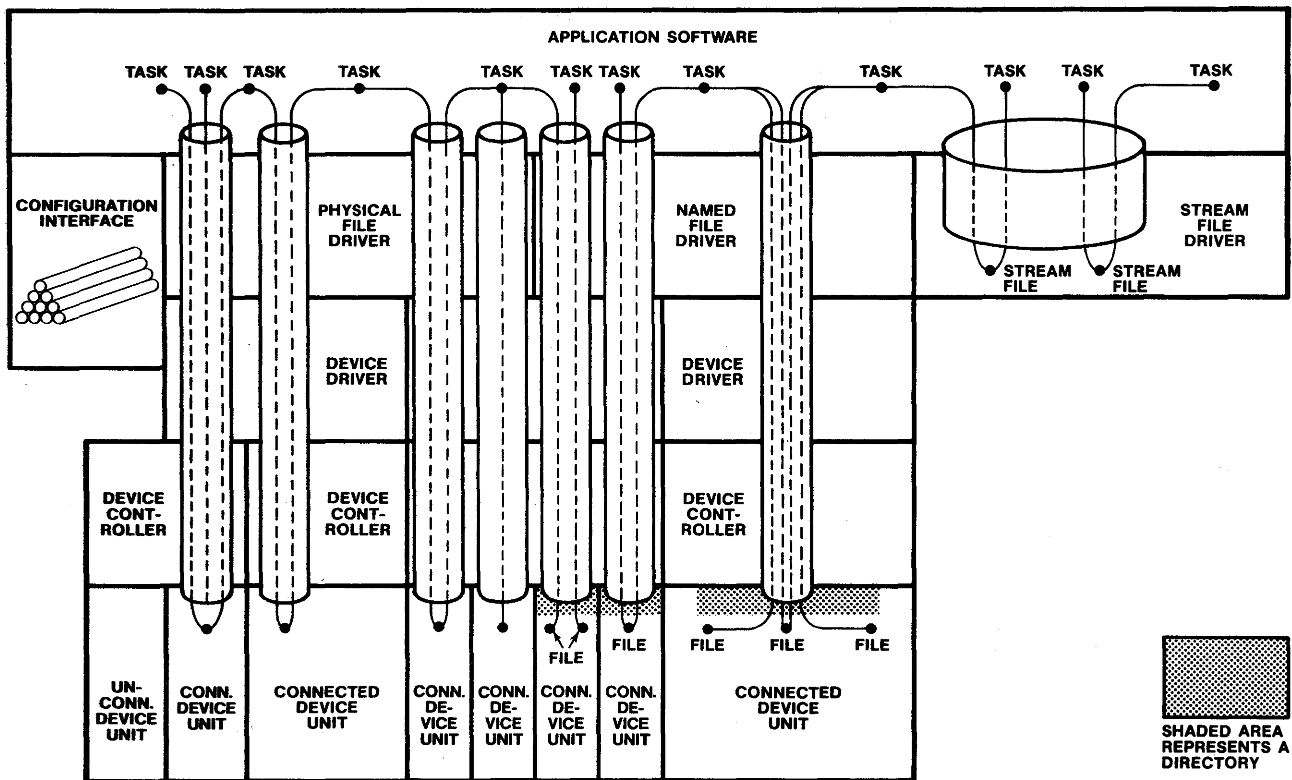


Figure 5-3. A System with Device and File Connections

FILE ACCESS LISTS

The access list for a file is a collection of up to three pairs of ID's and access masks. The ID's represent users or groups of users, and the access masks specify the kinds of access to the file that those users or groups of users are allowed.

Tasks calling CREATE$FILE pass an access mask and a token for a user object. The I/O System pairs the owner ID from the user object with the access mask and places the pair in the file's access list.

Tasks can alter the access list for a file by the CHANGE$ACCESS system call. Through CHANGE$ACCESS, ID-access pairs can be added or deleted, and the access masks for ID's already in the list can be changed.

ACCESS MASKS FOR FILE CONNECTIONS

When a task calls either CREATE$FILE or ATTACH$FILE, the I/O System constructs an access mask and binds it to the connection object returned by the call. After that, each time a task uses the connection object to try to access (open, close, read, write, etc.) the file, the I/O System checks the access mask to see if the kind of access being attempted is valid. Figure 5-4 illustrates the algorithm used to construct the access mask.

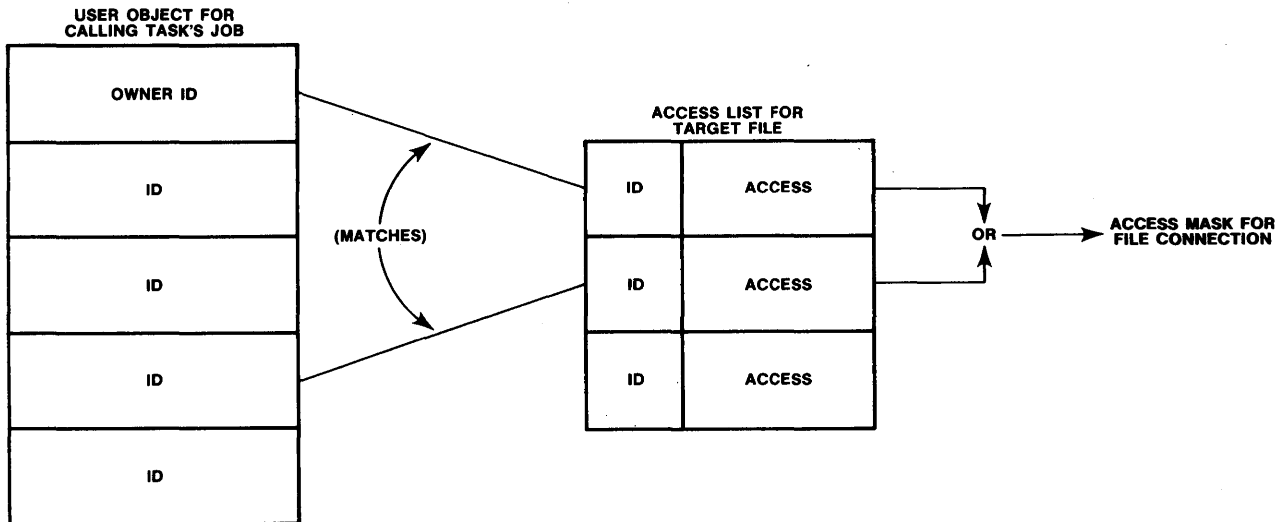


Figure 5-4. Computing the Access Mask for a File Connection

The I/O System compares the ID's in the specified user object with the ID's in the file's access list. The access masks corresponding to matching ID's are logically ORed, forming an aggregate mask.

EXTENDING A FILE DESCRIPTOR

For each named file on a disk, the I/O System creates and maintains a file descriptor on the same disk. The first portion of the descriptor contains information for the I/O System. The last three bytes are available to your operating system extensions, unless you are using the Human Interface, in which case only the last byte is available.

If you are writing an operating system extension and you want to record special information in a file's descriptor, SET$EXTENSION$DATA will place the data in the trailing portion of the descriptor. GET$EXTENSION$DATA can access this data when it is needed later.

NOTE

> If you are using the Human Interface, you must take care not to destroy the data the Human Interface keeps in the first two extra bytes of file descriptors. To preserve this data, first use GET$EXTENSION$DATA to read the data, next modify the third byte without disturbing the first two bytes, and finally use SET$EXTENSION$DATA to transfer the data to the descriptor.

CHAPTER 6. DELETION CONSIDERATIONS

In an iRMX 86-based system, tasks run asynchronously. Unless special precautions are taken, one task cannot know what another is doing.

This asynchronous behavior can lead to problems relating to the deletion of objects. Suppose that two tasks, A and B, both use a certain mailbox. If the tasks do not coordinate with each other, a sequence of events similar to the following might occur:

1) Task A sends a segment to the shared mailbox.

2) Task B, which was waiting at the mailbox, awakens and deletes the mailbox.

3) Task A, which is unaware that the mailbox has been deleted, tries to use the mailbox and incurs an exception.

Scenarios such as this can be avoided by good programming practices. If two tasks share an object, it is your responsibility to ensure that no task deletes an object until other tasks are finished using it.

CHAPTER 7. SYSTEM CALLS

This chapter contains the calling sequences and other information about advanced system calls to the Nucleus and I/O System. 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 system call 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.

SYSTEM CALL DICTIONARY

System Call	Synopsis	Page
Composite Objects		
ALTER$COMPOSITE	Alters the component list of a composite object	7-17
CREATE$COMPOSITE	Creates a composite object	7-19
DELETE$COMPOSITE	Deletes a composite object	7-26
INSPECT$COMPOSITE	Returns a list of the component object tokens contained in a composite object	7-36
Configuration Interface		
A$PHYSICAL$AT-TACH$DEVICE	Attaches a device to the Basic I/O System	7-8
A$PHYSICAL$DE-TACH$DEVICE	Detaches a device from the Basic I/O System	7-11
LOGICAL$ATTACH$-DEVICE	Attaches a device to the Extended I/O System	7-40
LOGICAL$DETACH$-DEVICE	Detaches a device from the Extended I/O System	7-43
Deletion Control		
DISABLE$DELETION	Increases the deletion disabling depth of an object by one	7-31
ENABLE$DELETION	Decreases the deletion disabling depth of an object by one	7-33
FORCE$DELETE	Forces the deletion of an objecteven if the object has had its deletion disabled once	7-34
Extension Objects		
CREATE$EXTENSION	Creates a new extension object type	7-21
DELETE$EXTENSION	Deletes an extension type	7-27

SYSTEM CALL DICTIONARY (continued)

System Call	Synopsis	Page
File Drivers		
AGETEXTENSION$DATA	Returns from the I/O System extension data stored with a file	7-5
ASETEXTENSION$DATA	Sets the extension data for a file from the I/O System	7-13
Operating System Extensions		
SETOSEXTENSION	Allocates and deallocates extension entries in the interrupt vector table	7-47
SIGNAL$EXCEPTION	Signals the occurrence of an exceptional condition	7-52
Priority Control		
SET$PRIORITY	Changes the priority of a task dynamically	7-49
Regions		
ACCEPT$CONTROL	Requests access to data protected by a region only if access is immediately available	7-16
CREATE$REGION	Creates a region	7-23
DELETE$REGION	Deletes a region	7-29
RECEIVE$CONTROL	Requests eventual access to data protected by a region	7-45
SEND$CONTROL	Relinquishes access to data protected by a region	7-46

SYSTEM CALL DICTIONARY (continued)

System Call	Synopsis	Page
Time		
SET$TIME	Sets the time and the date	7-51
User Objects		
CREATE$USER	Creates a user object	7-24
DELETE$USER	Deletes a specified user object	7-30
INSPECT$USER	Returns a list of the ID's in a user object	7-38

AGETEXTENSION$DATA

The AGETEXTENSION$DATA (Basic I/O) system call returns extension data stored with a Basic I/O System file.

```
CALL RQ$A$GET$EXTENSION$DATA(connection, resp$mbox, except$ptr);
```

INPUT PARAMETERS

connection	WORD containing a token of an asynchronous connection object for a file whose extension data is desired.
resp$mbox	WORD containing a token for the mailbox to which a segment is to be returned.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

Associated with each file created through the Basic I/O System is a file descriptor containing information about the file. Some of that information is used by the Basic I/O System and can be accessed by tasks through the AGETFILE$STATUS system call. Three additional bytes of the file descriptor, known as extension data, are available for use by Operating System extensions. OS extensions can write extension data by using ASETEXTENSION$DATA and they can read extension data by using AGETEXTENSION$DATA.

When a task calls AGETEXTENSION$DATA, it specifies a response mailbox to which the system returns a segment with the extension data. The information returned has the following form and is situated in the low-memory portion of the segment:

```
DECLARE
    ext$data$seg STRUCTURE(
         status  WORD
         count   BYTE,
         info(*) BYTE);
```

AGETEXTENSION$DATA (continued)

DESCRIPTION (continued)

Status indicates the success or failure of the operation. If status does not contain an E$OK condition code, then neither count nor info is valid. Count specifies the number (up to three) of bytes that are returned. Info contains the extension data.

AGETEXTENSION$DATA can only be applied to connections created via the named file driver.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The connection was created by the Extended I/O System and includes data buffers.
E$EXIST	Any of the following conditions exist: • One or both of the connection or resp$mbox arguments does not refer to an existing object. • The connection is in the process of being deleted.
E$IFDR	The get extension data request is not valid for files supported by the file driver implied in the call.
E$IO	An I/O error occurred during the operation.
E$LIMIT	The I/O System cannot create an I/O result segment because the calling task's job has already reached its object limit.
E$MEM	The I/O System cannot create an I/O result because the memory pool of the calling task's job does not have a sufficiently large block.
E$NOPREFIX	No default prefix has been set.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$NOUSER	Either: • No default user has been set. • The default user cataloged is not a user object

CONDITION CODES (continued)

E$SUPPORT	The connection was created by a task whose job is different than the calling task's job.
E$TYPE	One or both of the connection or resp$mbox arguments is not of the correct object type.

A$PHYSICAL$ATTACH$DEVICE

The A$PHYSICAL$ATTACH$DEVICE (Basic I/O) system call attaches a device to the Basic I/O System.

NOTE

Any task invoking this call must have a priority in the range 32 to 255.

```
CALL RQ$A$PHYSICAL$ATTACH$DEVICE(dev$name, file$driver, resp$mbox,
                                 except$ptr);
```

INPUT PARAMETERS

dev$name — POINTER to a STRING containing the name (as specified during configuration) of the device to be attached.

file$driver — BYTE specifying which file driver is to supply the connection to the device. Possible values are as follows:

Value	File Driver
1	Physical
2	Stream
4	Named

resp$mbox — WORD containing a token for the mailbox to which the results of this call will be returned.

OUTPUT PARAMETER

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

A$PHYSICAL$ATTACH$DEVICE creates a connection to a physical or logical device. Such a device connection must be in place before any file connections to files on the device can be created.

DESCRIPTION (continued)

A connection object is returned to the response mailbox if the call is successful; otherwise an I/O result segment is returned to the response mailbox. The device connection object returned can be used as a prefix in other system calls. It can be deleted only by calling the A$PHYSICAL$DETACH$DEVICE system call.

In the case of connections to disk devices, the connection is actually to a volume mounted on the disk hardware. Such volumes must be properly formatted. If they are not, an E$ILLVOL exceptional condition is returned. Refer to the iRMX 86 INSTALLATION GUIDE or the iRMX 86 HUMAN INTERFACE REFERENCE MANUAL for information regarding the formatting of disks.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This code is returned in the result segment to indicate that the specified device is already attached.
E$DEVFD	The specified device is not compatible with the specified file driver.
E$EXIST	The resp$mbox argument does not refer to an existing object.
E$FNEXIST	The device specified by the device$name parameter does not exist.
E$ILLVOL	The specified device is a disk volume not properly formatted for use with the named file driver.
E$IO	An I/O error occured during the operation.
E$LIMIT	If this code is returned synchronously, the Basic I/O System attempted to create an object and the calling task's job has already reached its object limit. If returned in the result segment, this code indicates that the Basic I/O System job has already reached its object limit.
E$MEM	If this code is returned synchronously, The Basic I/O System cannot create an I/O result segment because the memory pool of the calling task's job does not have a sufficiently large block. If returned in the result segment, this code indicates that the Basic I/O System job does not have a sufficiently large block of memory.

A$PHYSICAL$ATTACH$DEVICE (continued)

CONDITION CODES (continued)

E$PARAM	The number representing the file driver is not valid.
E$TYPE	The resp$mbox argument contains a token for an object that is not mailbox.

A$PHYSICAL$DETACH$DEVICE

The A$PHYSICAL$DETACH$DEVICE (Basic I/O) system call detaches a device from the Basic I/O System.

```
CALL RQ$A$PHYSICAL$DETACH$DEVICE(connection, hard, resp$mbox,
                                  except$ptr);
```

INPUT PARAMETERS

connection — WORD containing a token for the connection object for the device that is to be detached.

hard — BYTE containing a value which specifies whether (OFFH) or not (0) a hard detach of the device is desired.

resp$mbox — WORD containing a token for the mailbox to which the result segment is sent when the operation has finished. A value of zero indicates that no response is desired.

OUTPUT PARAMETER

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The A$PHYSICAL$DETACH$DEVICE breaks connections established by calls to A$PHYSICAL$ATTACH$DEVICE. It also deletes the file connection objects associated with those device connections. Devices that re-detached in this manner must be reattached before any files on the device can be attached.

When detaching a device, you can choose to detach all attached files on the device. A hard detach deletes the connection objects for all such files on the device. To specify a hard detach, assign the value OFFH to the hard parameter.

If you choose not to request a hard detach, there must not be any attached files on the device. To specify that you do not want a hard detach, assign the value 0 to the hard parameter.

A$PHYSICAL$DETACH$DEVICE (continued)

DESCRIPTION (continued)

Note that, whether or not you specify a hard detach, there will be no attached files on the device after the device is detached.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This code is returned synchronously to indicate either of the following: • The connection argument is for a connection other than to a device. • The connection was created by the Extended I/O System and includes data buffers.
E$EXIST	Either: • One or both of the connection or resp$mbox arguments does not refer to an existing object. • The connection is in the process of being deleted.
E$IFDR	This system call is not valid for the file driver associated with the device.
E$IO	An I/O error occurred during the operation.
E$LIMIT	The I/O System cannot create an I/O result segment but the calling task's job has already reached its object limit.
E$MEM	The I/O System cannot create an I/O result segment because the memory pool of the calling task's job does not have a sufficiently large block.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$NOUSER	The default user cataloged is not a user object.
E$SUPPORT	The task that created the connection is not in the same job as the calling task.
E$TYPE	One or both of the connection or resp$mbox arguments is not of the correct object type.

ASETEXTENSION$DATA

The ASETEXTENSION$DATA (Basic I/O) system call writes the extension data for a Basic I/O System file.

```
CALL RQ$A$SET$EXTENSION$DATA(connection, data$ptr, resp$mbox,
                             except$ptr);
```

INPUT PARAMETERS

connection — WORD containing a token for an asynchronous connection to a file whose extension data is to be set.

data$ptr — POINTER to the structure of the following form:

```
DECLARE ext$data$seg  STRUCTURE(
            count      BYTE,
            info(*)   BYTE);
```

where:

- count — Number (up to three) of bytes of extension data being written.
- info(*) — The extension data.

resp$mbox — WORD containing a token for the mailbox to receive the I/O result segment when the operation is finished. A value of zero indicates that no response is desired.

OUTPUT PARAMETER

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

Associated with each file created through the Basic I/O System is a file descriptor containing information about the file. Some of that information is used by the Basic I/O SYstem and can be accessed by tasks through the AGETFILE$STATUS system call. Three additional bytes of the file descriptor, known as extension data, are available for use by

ASETEXTENSION$DATA (continued)

DESCRIPTION (continued)

Operating System extensions. OS extensions can write extension data by using ASETEXTENSION$DATA and they can read extension data by using AGETEXTENSION DATA.

NOTE

If your system includes the Human Interface, then only the last byte of the extension data is available for use by your OS extenions. Take care, when using ASETEXTENSION$DATA, to preserve the contents of the first two bytes. Do this by calling AGETEXTENSION$DATA before writing into the third byte.

After the new extension data is set, an I/O result segment returns to the response mailbox.

ASETEXTENSION$DATA can only be applied to asynchronous connections created via the named file driver.

CONDITION CODES

E$OK — No exceptional conditions.

E$CONTEXT — The connection was created by the Extended I/O System and includes data buffers.

E$EXIST — Either:

- One or both of the connection or resp$mbox arguments does not refer to an existing object.
- The connection is in the process of being deleted.

E$IFDR — The set extension data request is not valid for files supported by the file driver implied in the call.

E$IO — An I/O error occurred during the operation.

E$LIMIT — The I/O System cannot create an I/O result segment because the calling task's job has already reached its object limit.

CONDITION CODES (continued)

E$MEM	The I/O System cannot create an I/O result segment because the memory pool of the calling task's job does not have a sufficiently large block.
E$NOPREFIX	No default prefix has been set.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$NOUSER	Either: • No default user has been set. • The default user cataloged is not a user object.
E$PARAM	The count field in the data structure contains a value greater than three.
E$SUPPORT	The task that created the connection was not in the same job as the calling task.
E$TYPE	One or both of the connection or resp$mbox arguments is not of the correct object type.

ACCEPT$CONTROL

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

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

INPUT PARAMETER

region	WORD containing a token for the target region.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CONDITION CODES

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

ALTER$COMPOSITE

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

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

INPUT PARAMETERS

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

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

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

ALTER$COMPOSITE (continued)

CONDITION CODES

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

The CREATE$COMPOSITE (Nucleus) system call creates a composite object.

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

INPUT PARAMETERS

extension — WORD containing a token for an extension type representing license to create a composite object.

token$list — POINTER to a structure of the form:

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

where:

num$slots — Number of positions available for tokens in token$list.

num$used — Number of component tokens making up the composite object.

token(*) — Tokens that will actually constitute the composite object.

OUTPUT PARAMETERS

composite — WORD where a token for the new composite object will be returned.

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CREATE$COMPOSITE (continued)

DESCRIPTION (continued)

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

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

CONDITION CODES

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

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

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

INPUT PARAMETERS

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

OUTPUT PARAMETERS

extension	WORD where a token for the new type will be returned.
except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

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

CREATE$EXTENSION (continued)

DESCRIPTION (continued)

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

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

CONDITION CODES

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

The CREATE$REGION (Nucleus) system call creates a region.

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

INPUT PARAMETER

region$flags	WORD which if the low order bit equals zero, tasks await access in FIFO order; if the low order bit equals one, tasks await access in priority order. The other bits in the WORD are reserved and should be set to zero.

OUTPUT PARAMETERS

region	WORD where a token for the newly created region will be returned.
except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CONDITION CODES

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

CREATE$USER

The CREATE$USER (Basic I/O) system call creates a user object.

```
user=RQ$CREATE$USER(ids$ptr, except$ptr);
```

INPUT PARAMETER

ids$ptr — a POINTER to a structure of the following form:

```
DECLARE  ids STRUCTURE(
    length      WORD,
    count       WORD,
    ID(*)       WORD);
```

where:

length — Number of elements in the ID array.

count — Number of ID's in the ID array that are to be included in the user object. This number must be less than or equal to length, but greater than or equal to one.

ID — Array of ID's, each of which is included in the user object. The first ID is to be used as the owner ID for any file created with reference to this user object.

OUTPUT PARAMETERS

user — WORD where a token for the new user object will be returned.

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The CREATE$USER system call creates a user object. It accepts a list of ID's and returns a token for the new object.

If the number of ID slots, as specified by the length field, is greater than the number of ID's, as specified by the count field, the effect is as if length had been reduced to equal count.

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	The calling task's job has already reached its object limit.
E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block to satisfy the request.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$PARAM	The count field in the ids structure is either zero or greater than the length field.

DELETE$COMPOSITE

The DELETE$COMPOSITE (Nucleus) system call deletes a composite object.

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

INPUT PARAMETERS

extension	WORD containing a token for the extension type used as a license to create the composite object to be deleted.
composite	WORD containing a token for the composite object to be deleted.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CONDITION CODES

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

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

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

INPUT PARAMETER

extension	WORD containing a token for the extension object to be deleted.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

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

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

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

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The extension argument does not refer to an existing object.

DELETE$EXTENSION (continued)

CONDITION CODES (continued)

E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block for Nucleus housekeeping purposes.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$TYPE	The extension argument does not contain a token for an extension object.

The DELETE$REGION (Nucleus) system call deletes a region.

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

INPUT PARAMETER

region	WORD containing a token for the region to be deleted.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CONDITION CODES

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

DELETE$USER

The DELETE$USER (Basic I/O) system call deletes a user object.

```
CALL RQ$DELETE$USER(user, except$ptr);
```

INPUT PARAMETER

user	WORD containing a token for the user object to be deleted.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The DELETE$USER system call deletes a user object. Deleting a user object has no effect on connections created with the user object.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The user argument does not refer to an existing object.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$TYPE	The user argument does not contain a token for a user object.

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

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

INPUT PARAMETER

object	WORD containing a token for the object whose deletion is to be disabled.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

NOTES

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

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

CONDITION CODES

E$OK	No exceptional conditions.

DISABLE$DELETEION (continued)

CONDITION CODES (continued)

E$EXIST	The object argument does not refer to an existing object.
E$LIMIT	The object's disabling depth is already 255.
ENOTCONFIGURED	This system call is not part of the present configuration.

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

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

INPUT PARAMETER

object	WORD containing a token for the object whose deletion is to be enabled.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The object's deletion is not disabled.
E$EXIST	The object argument does not refer to an existing object.
ENOTCONFIGURED	This system call is not part of the present configuration.

FORCE$DELETE

The FORCE$DELETE (Nucleus) system call deletes objects whose disabling depths are zero or one.

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

INPUT PARAMETERS

extension	If the object to be deleted is a composite object, this parameter is a WORD containing a token for the extension type associated with the composite object to be deleted. Otherwise, the extension argument must be zero.
object	WORD containing a token for the object that is to be deleted.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The FORCE$DELETE system call deletes objects whose disabling depths are zero or one. If an object has a deletion depth of two or more, the calling task is put to sleep until the deletion depth is decreased to one. At that time, the object is deleted and the task is awakened.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	One or both of the object or extension arguments does not refer to an existing object.
E$MEM	The memory pool of the calling task's job does not contain a sufficiently large block for Nucleus housekeeping purposes.

CONDITION CODES (continued)

ENOTCONFIGURED	This system call is not part of the present configuration.
E$TYPE	The extension argument does not contain a token for an extension type.

INSPECT$COMPOSITE

The INSPECT$COMPOSITE (Nucleus) system call returns a list of the component tokens contained in a composite object.

```
CALL RQ$INSPECT$COMPOSITE(extension, composite, token$list,
                          except$ptr);
```

INPUT PARAMETERS

extension — WORD containing a token for the extension object corresponding to the composite object being inspected.

composite — WORD containing a token for the composite object being inspected.

OUTPUT PARAMETERS

token$list — POINTER to a structure of the form:

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

where:

num$slots — Number of positions available for tokens in token$list (an upper limit on the number of tokens to be returned).

num$used — Number of component tokens making up the composite object.

token(*) — The tokens that actually constitute the composite object.

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The INSPECT$COMPOSITE system call accepts a token for a composite object and returns a list of tokens for the components of the composite object.

The calling task must supply the num$slots value in the data structure pointed to by the token$list parameter. The Nucleus fills in the remaining fields in that structure. If num$slots is set to zero, the Nucleus will fill in only the num$used field.

If the num$slots value is smaller than the actual number of component tokens, only that number (num$slots) of tokens will be returned.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The composite argument is not compatible with the extension argument.
E$EXIST	One or both of the extension or composite arguments does not refer to a currently existing object.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$TYPE	One or both of the extension or composite arguments is not of the correct object type.

INSPECT$USER

The INSPECT$USER (Basic I/O) System call returns a list of the ID's contained in a user object.

```
CALL RQ$INSPECT$USER(user, ids$ptr, except$ptr);
```

INPUT PARAMETER

user — WORD containing a token for the user object being inspected.

OUTPUT PARAMETERS

ids$ptr — POINTER to a structure of the following form:

```
DECLARE  ids  STRUCTURE(
    length        WORD,
    count         WORD,
    id(*)         WORD);
```

where:

length — Upper limit on the number of ID's that are to be returned.

count — Actual number of ID's that are being returned.

id(*) — The ID's being returned.

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The INSPECT$USER system accepts a token for a user object and returns a list of the ID's in the user object.

The calling task must supply the length value in the data structure pointed to by the ids$ptr parameter. The I/O System fills in the remaining fields in that structure.

If the length value is smaller than the actual number of ID's in the user object, only the specified number of ID's will be returned.

CONDTION CODES

E$OK	No exceptional conditions.
E$EXIST	The user argument does not refer to an existing object.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$PARAM	The length field contains a value of zero.
E$TYPE	The user argument contains a value that is not a token for a user object.

LOGICAL$ATTACH$DEVICE

The LOGICAL$ATTACH$DEVICE (Extended I/O) system call assigns a logical name to a physical device. It does this by creating a Logical Device Object and cataloging it under the specified logical name in the root object directory.

```
CALL RQ$LOGICAL$ATTACH$DEVICE(log$name, dev$name, file$driver,
                              except$ptr);
```

INPUT PARAMETERS

log$name — POINTER to a STRING containing the logical name under which the logical device object is to be cataloged.

dev$name — POINTER to a STRING containing the name (as specified in the DUIB during Basic I/O System configuration) of the device to be assigned.

file$driver — BYTE specifying which I/O System file driver is to use the device. Possible values are as follows:

value	file driver
1	physical
2	stream
4	named

OUTPUT PARAMETER

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

LOGICAL$ATTACH$DEVICE creates a Logical Device Object corresponding to a physical device and then catalogs the object in the root object directory under the logical name specified in the call. Such a logical device object must be in place before any file connections to files on the device can be created. The Extended I/O System attaches the physical device (creates a device connection) during the first Extended I/O System call that uses this logical name as the prefix of a path name. The logical name can be used as a prefix in other system calls and can be deleted by LOGICAL$DETACH$DEVICE.

DESCRIPTION (continued)

Because of the nature of LOGICAL$ATTACH$DEVICE, some execption codes that result because of errors in this system call will not be returned until the Extended I/O System actually tries to attach the device (during the first system call that uses the logical name as the prefix of a path name).

CONDITION CODES

E$OK — No exceptional conditions.

E$CONTEXT — The root object directory already contains an entry with the name pointed to by the log$name parameter.

E$LIMIT — Either:

- The root object directory is full.
- The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information concerning I/O jobs.

E$MEM — The memory pool of the calling task's job does not have a sufficiently large block of memory to allow this system call to run to completion.

ENOTCONFIGURED — At least one of the following system calls was left out during the configuration process:

CATALOG$OBJECT (Nucleus)
CREATE$COMPOSITE (Nucleus)
CREATE$MAILBOX (Nucleus)
CREATE$SEGMENT (Nucleus)
GET$TYPE (Nucleus)
LOGICAL$ATTACH$DEVICE (Extended I/O System)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
SEND$MESSAGE (Nucleus)

E$PARAM — This code indicates that the specified logical name is syntactically incorrect. Any one of the following problems can cause this error:

- The STRING pointed to by the log$name parameter is of zero length.

LOGICAL$ATTACH$DEVICE (continued)

CONDITION CODES
E$PARAM (continued)

- The STRING pointed to by the log$name parameter has a length of greater than 12.
- The logical name contains invalid characters.

The LOGICAL$DETACH$DEVICE (Extended I/O) system call removes the correspondence between a logical name and a physical device that was established with the LOGICAL$ATTACH$DEVICE system call. It removes the logical name from the root object directory.

```
CALL RQ$LOGICAL$DETACH$DEVICE(log$name, except$ptr);
```

INPUT PARAMETER

log$name	POINTER to a STRING containing the name under which the logical device object is cataloged in the root object directory.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

LOGICAL$DETACH$DEVICE severs the association created by a call to LOGICAL$ATTACH$DEVICE and deletes the corresponding entry in the root object directory. At this point the device is logically detached; users cannot create new connections using the logical name as a prefix. When the last file connection on the physical device is severed, the Extended I/O System detaches the device (deletes the device connection). A task can then reassign the device.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The device connection corresponding to this logical name is in the process of being deleted.

LOGICAL$DETACH$DEVICE (continued)

CONDITION CODES (continued)

E$LIMIT — This condition code can be caused by either of the following conditions:

- Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations.
- The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information concerning I/O jobs.

ELOGNAME$NEXIST — The logical name was not found in the root object directory.

E$MEM — The memory pool of the calling task's job does not have a sufficiently large block of memory to allow this system call to run to completion.

ENOTCONFIGURED — At least one of the following system calls was left out during the configuration process:

A$PHYSICAL$DETACH$DEVICE (Basic I/O System)
CREATE$MAILBOX (Nucleus)
CREATE$SEGMENT (Nucleus)
DELETE$COMPOSITE (Nucleus)
GET$TYPE (Nucleus)
LOGICAL$DETACH$DEVICE (Extended I/O System)
LOOKUP$OBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
SEND$CONTROL (Nucleus)
UNCATALOG$OBJECT (Nucleus)

ENOTDEVICE — When the Extended I/O System looked up the token associated with the logical name, it was not a valid device connection.

E$PARAM — This code indicates that the specified logical name is syntactically incorrect. Any one of the following problems can cause this error:

- The STRING pointed to by the log$name parameter is of zero length.
- The STRING pointed to by the log$name parameter has a length of greater than 12.
- The logical name contains invalid characters.

The RECEIVE$CONTROL (Nucleus) system call allows the calling task to gain access to data protected by a region.

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

INPUT PARAMETER

region	WORD containing a token for the region protecting the data to which the calling task wants access.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The RECEIVE$CONTROL system call requests access to data protected by a region. If no task currently has access, entry is immediate. If another task currently has access, the calling task is placed in the region's task queue and goes to sleep. The task remains asleep until it gains access to the data.

If the region has a priority-based task queue, the priority of the task currently having access is temporarily boosted, if necessary, to match that of the task at the head of the queue.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The region argument refers to a region already accessed by the calling task.
E$EXIST	The region argument does not refer to an existing object.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$TYPE	The region argument does not contain a token for a region.

SEND$CONTROL

The SEND$CONTROL (Nucleus) system call allows a task to surrender access to data protected by a region.

```
CALL RQ$SEND$CONTROL(except$ptr);
```

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

When a task finishes with data protected by a region, it invokes the SEND$CONTROL system call to surrender access. If the task is using more than one set of data, each of which is protected by a region, the SEND$CONTROL system call surrenders the most recently obtained access. When access is surrendered, the system allows the next task in line to gain access.

If a task calling SEND$CONTROL has had its priority boosted while it had access through a region, its priority is restored when it relinquishes the access.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	A task invoked the SEND$CONTROL while it did not have access to data protected by any region.
ENOTCONFIGURED	This system call is not part of the present configuration.

The SETOSEXTENSION (Nucleus) system call either enters the address of an entry (or function) procedure in the interrupt vector table or it deletes such an entry.

```
CALL RQ$SET$OS$EXTENSION(os$extension, start$address, except$ptr);
```

INPUT PARAMETERS

os$extension	BYTE designating the entry of the interrupt vector table to be set or reset. This value must be between 224 and 255 (decimal), inclusive (the values in the range 192 to 223 are valid, but are reserved for Intel use).
start$address	POINTER to the first instruction of an entry (or function) procedure. If start$address contains a zero value, the specified interrupt vector table entry is being reset (deallocated).

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The SETOSEXTENSION system call sets or resets any one of the 32 operating system extension entries in the interrupt vector. An entry must be reset before its contents can be changed. An attempt to set an already set entry causes an E$CONTEXT exceptional condition.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	An attempt is being made to set an entry that already is set.

SETOSEXTENSION (continued)

CONDITION CODES (continued)

ENOTCONFIGURED	This system call is not part of the present configuration.
E$PARAM	The OS$extension byte value is less than 192.

The SET$PRIORITY (Nucleus) system call changes the priority of a task.

```
CALL RQ$SET$PRIORITY(task, priority, except$ptr);
```

INPUT PARAMETERS

task	WORD containing a token for the task whose priority is to be changed. A zero value specifies the invoking task.
priority	BYTE containing the task's new priority. A zero value specifies the maximum priority of the specified task's containing job.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The SET$PRIORITY system call allows the priority of a noninterrupt task to be altered dynamically.

If the priority parameter is set to the zero, the task's new priority is its containing job's maximum priority. Otherwise, the priority parameter contains the new priority of the specified task. The new priority, if explicitly specified, must not exceed its containing job's maximum priority.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	An attempt is being made to change the priority of an interrupt task.
E$EXIST	The task argument does not refer to an existing object.

SET$PRIORITY (continued)

CONDITION CODES (continued)

E$LIMIT	The priority parameter contains a priority value that is higher than the maximum priority of the specified task's containing job.
ENOTCONFIGURED	This system call is not part of the present configuration.
E$TYPE	The task argument does not contain a token for a task.

The SET$TIME (Basic I/O) system call sets the date and time for the I/O System.

```
CALL RQ$SET$TIME(time$high, time$low, except$ptr);
```

INPUT PARAMETERS

time$high	WORD specifying the first half (the most significant 16 bits) of the value of the date and time.
time$low	WORD specifying the second half (the least significant 16 bits) of the value of the date and time.

OUTPUT PARAMETER

except$ptr	POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The SET$TIME system call sets the date/time value for the I/O system. The I/O System maintains the date/time value as two words containing the number of seconds since a fixed point in time. Any time in the past can be used as the "beginning of time", but we recommend that you use 12:00 am (midnight), January 1, 1978.

CONDITION CODES

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

SIGNAL$EXCEPTION

The SIGNAL$EXCEPTION (Nucleus) system call is invoked by OS extensions to signal the occurrence of an exceptional condition.

```
CALL RQ$SIGNAL$EXCEPTION(exception$code, param$num, stack$pointer,
                         reserved$word, reserved$word, except$ptr);
```

INPUT PARAMETERS

exception$code — WORD containing the code (see list in Appendix B) for the exceptional condition detected.

param$num — BYTE containing the number of the parameter which caused the exceptional condition. If no parameter is at fault, param$num equals zero.

stack$pointer — WORD which, if not zero, must contain the value of the stack pointer saved on entry to the operating system extension (see the entry procedure in Chapter 4 for an example). The top five words in the stack (where BP is at the top of the stack) must be as follows:

FLAGS	Saved by software interrupt
CS	to OS extension
IP	
DS	Saved by OS extension
BP	on entry

Upon completion of SIGNAL$EXCEPTION, control returns to the instruction identified in CS and IP.

If stack$pointer contains a zero, control returns, upon completion of SIGNAL$EXCEPTION, to the instruction following the call to SIGNAL$EXCEPTION.

reserved$word — Two WORDs reserved for Intel use. Set these parameters to zero.

OUTPUT PARAMETER

except$ptr — POINTER to a WORD where the condition code will be returned.

DESCRIPTION

Operating system extensions use the SIGNAL$EXCEPTION system call to signal the occurrence of exceptional conditions. Depending on the exceptional condition and the calling task's exception mode, control may or may not pass directly to the task's exception handler.

If the exception handler does not get control, the exceptional condition code is returned to the calling task. The task can then access the code by checking the contents of the word pointed to by the except$ptr argument for its call (not for the call to SIGNAL$EXCEPTION).

CONDITION CODES

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

APPENDIX A. iRMX 86™ DATA TYPES

The following are the data types that are recognized by the iRMX 86 Operating System:

BYTE - An unsigned, one byte, 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 the iRMX 86 system by means of a numeric code. For each code, there is a mnemonic name that can be substituted for the code. The following lists the types with their codes and associated mnemonics.

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 0FFFH, depending on the value specified in CREATE$EXTENSION
User (in Basic I/O System)	TNUMUSER	100H
Connection (in Basic I/O System)	TACONNECTION	101H
I/O Job (in Extended I/O System)	TIOJOB	300H
Logical Device Object (in Extended I/O System)	TLOGDEV	301H

INDEX

For most topics with multiple-page references, the primary reference is underscored.

ACCEPT$CONTROL 2-4, 2-6, 7-16
access list 5-7
access mask 5-7
air-traffic-control application 2-1
ALTER$COMPOSITE 4-2, 4-6, 4-17, 7-17
application programmer 1-1, 2-5
ATTACH$FILE 5-7
attaching devices 5-1
AX register 3-6

changing task priority 7-49
component token list 7-19, 7-36
composite object 4-1, 7-17, 7-19, 7-26, 7-36
configuration interface 5-1
CREATE$COMPOSITE 4-1, 4-12, 4-17, 7-19
CREATE$EXTENSION 4-1, 4-3, 4-17, 7-21
CREATE$FILE 5-7
CREATE$REGION 2-6, 7-23
CREATE$RING$BUFFER procedure 4-12
CREATE$USER 5-5, 7-24
custom object types 4-1
CX register 3-6, 3-8

deadlock 2-4
default user object 5-5
DELETE$COMPOSITE 4-2, 4-4, 4-17, 7-26
DELETE$EXTENSION 4-2, 4-5, 4-17, 7-27
DELETE$JOB 4-3
DELETE$REGION 2-6, 7-29
DELETE$RING$BUFFER procedure 4-14
DELETE$USER 5-5, 7-30
deletion considerations 6-1
deletion mailbox 4-2, 7-21, 7-27
deletion prevention 2-3, 3-14
detaching devices 5-1
device connection 5-4, 7-8, 7-11, 7-40, 7-43,
device driver 5-2
device unit 5-2
dictionary of system calls 7-2
DISABLE$DELETION 3-14, 7-31,
disabling depth 3-14, 7-31, 7-33, 7-34
DL register 3-6, 3-8

ENABLE$DELETION 3-14, 7-33
entry procedure 3-3, 3-7, 7-47
exception handler 3-7, 3-11, 7-52
exceptional conditions 3-6, 3-8, 3-10
extension data 5-8, 7-5, 7-13

file access 5-5
file descriptor 5-8
file driver 5-2
file protection 5-7
FORCE$DELETE 3-15, 7-32
function procedure 3-3, 3-10, 7-47

GET$BYTE procedure 4-15
GET$EXCEPTION$HANDLER 3-7
GET$EXTENSION$DATA 5-8, 7-5

hard detach 7-11

I/O System hardware 5-1
ID 5-5, 7-24, 7-38
in-line exception handling 3-8, 3-12
initialization part 4-6, 4-8
INSPECT$COMPOSITE 4-2, 4-12, 7-36
INSPECT$USER 5-5, 7-38
inspecting composite objects 4-2, 4-12, 7-36
interface library 3-7, 4-9
interface procedure 3-3, 3-6, 4-9
interrupt vector 3-3, 7-47
interupt vector table 7-47
intertask coordination 2-1

language interface 3-3
linking procedures 3-7
logical
    names 5-1, 5-3, 7-40, 7-43
    device object 5-3, 7-40, 7-43
LOGICAL$ATTACH$DEVICE 5-1, 5-3, 7-40
LOGICAL$DETACH$DEVICE 5-1, 5-3, 7-43

manipulating composite objects and extension types 4-2
mutual exclusion 2-2, 2-3

named file driver 5-3, 5-6, 7-8
nested composites 4-5
nesting regions 2-4

operating system extension 3-1, 4-1
OS extension 3-1, 4-1

physical file driver 5-2, 5-6, 7-8
PHYSICAL$ATTACH$DEVICE 5-1, 5-3, 7-8
PHYSICAL$DETACH$DEVICE 5-1, 5-3, 7-11
priority adjustment 7-49
priority boosting 2-3

priority bottlenecks 2-2
procedure libraries 3-1
protection against deletion 3-14
PUT$BYTE procedure 4-14

RECEIVE$CONTROL 2-4, 2-6, 7-45
region 2-1
ring buffer 4-7
ring buffer example 4-7
ring buffer manager 4-7
RQ$ERROR procedure 3-6, 3-10

semaphores 2-2, 2-4
SEND$CONTROL 2-6, 7-46
SET$EXCEPTION$HANDLER 3-7
SET$EXTENSION$DATA 5-8, 7-13
SETOSEXTENSION 3-14, 3-15, 7-47
SET$PRIORITY 7-49
SET$TIME 7-51
shared data 2-1, 7-16, 7-45, 7-46
SIGNAL$EXCEPTION 3-6, 3-10, 3-12, 7-52
signalling exceptions 3-6, 3-8, 3-10, 7-52
stream file driver 5-2, 5-6, 7-8
system call dictionary 7-2
system programmer 1-1

time 7-51
type code 4-1, 7-21
type manager 4-1

user object 5-5, 7-24, 7-30, 7-38
user-supplied operating system extensions 3-1, 4-1

verify access to files 5-7

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 ______________________

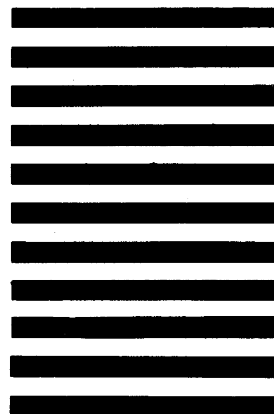
CITY ______________________ STATE ________ ZIP CODE __________

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, California 95051 (408) **987-8080**

Printed in U.S.A.