intel®

EXTENDED iRMX® II.3 OPERATING SYSTEM DOCUMENTATION

VOLUME 2 USER'S GUIDES

Order Number: 461845-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

Above	iLBX	iPSC	OpenNET
BITBUS	i_m	iRMX	ONCE
COMMputer	iMDDX	iSBC	Plug-A-Bubble
CREDIT	iMMX	iSBX	PROMPT
Data Pipeline	Insite	iSDM	Promware
Genius	int$_e$l	iSSB	QUEST
$\overset{\Delta}{\text{i}}$	int$_e$lBOS	iSXM	QueX
i	Intelevision	Library Manager	Ripplemode
I^2ICE	int$_e$ligent Identifier	MCS	RMX/80
ICE	int$_e$ligent Programming	Megachassis	RUPI
iCEL	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SLD
iDBP	iOSP	MULTICHANNEL	UPI
iDIS	iPDS	MULTIMODULE	VLSiCEL
	iPSB		

MANUALS IN THIS VOLUME

This volume (Volume 2, *Extended iRMX® II* User's Guides) contains the following manuals, all of which document the iRMX II layers. These manual are intended to provide the information needed to use the iRMX II Operating System.

Extended iRMX® II Nucleus User's Guide
Extended iRMX® II Basic I/O System User's Guide
Extended iRMX® II Extended I/O System User's Guide
Extended iRMX® II Application Loader User's Guide
Extended iRMX® II Human Interface User's Guide
Extended iRMX® II UDI User's Guide
Extended iRMX® II Device Drivers User's Guide

The *Extended iRMX® II Nucleus User's Guide* describes the concepts of the innermost layer, the Nucleus.

The *Extended iRMX® II Basic I/O System User's Guide* describes the concepts of the Basic I/O System.

The *Extended iRMX® II Extended I/O System User's Guide* describes the concepts of the Extended I/O system.

The *Extended iRMX® II Application Loader User's Guide* describes how to use the Loader to load your programs.

The *Extended iRMX® II Human Interface User's Guide* explains how to use and modify the Human Interface.

The *Extended iRMX® II UDI User's Guide* describes the use of all UDI, the language interface.

The *Extended iRMX®* II Device Drivers User's Guide describes the data structures and support routines needed to write device drivers.

VOLUME CONTENTS

Manuals are listed in the order they appear in the volumes. For a synopsis of each manual, refer to the *Introduction to the Extended iRMX\ II Operating System*.

VOLUME 1: *Extended iRMX® II Introduction, Installation, and Operating Instructions*

Introduction to the Extended iRMX II Operating System
Extended iRMX II Hardware and Software Installation Guide
Operator's Guide to the Extended iRMX II Human Interface
Master Index

VOLUME 2: *Extended iRMX® II Operating System User Guides*

Extended iRMX® II Nucleus User's Guide
Extended iRMX® II Basic I/O System User's Guide
Extended iRMX® II Extended I/O System User's Guide
Extended iRMX® II Human Interface User's Guide
Extended iRMX® II Application Loader User's Guide
Extended iRMX® II Universal Development Interface User's Guide
Extended iRMX® II Device Drivers User's Guide

VOLUME 3: *Extended iRMX® II System Calls*

Extended iRMX® II Nucleus System Calls Reference Manual
Extended iRMX® II Basic I/O System Calls Reference Manual
Extended iRMX® II Extended I/O System Calls Reference Manual
Extended iRMX® II Application Loader System Calls Reference Manual
Extended iRMX® II Human Interface System Calls Reference Manual
Extended iRMX® II UDI System Calls Reference Manual

VOLUME 4: *Extended iRMX® II Operating System Utilities*

Extended iRMX® II Bootstrap Loader Reference Manual
Extended iRMX® II System Debugger Reference Manual
Extended iRMX® II Disk Verification Utility Reference Manual
Extended iRMX® II Programming Techniques Reference Manual
Guide to the Extended iRMX® II Interactive Configuration Utility

VOLUME 5: *Extended iRMX® II Interactive Configuration Utility Reference*

Extended iRMX® II Interactive Configuration Utility Reference Manual

REV.	REVISION HISTORY	DATE
-001	Original Issue.	01/88

EXTENDED iRMX® II NUCLEUS USER'S GUIDE

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual documents the Extended iRMX II Nucleus subsystem. The material contained in this manual is primarily intended for programmers who need to access system capabilities.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts contained in the *Introduction to the Extended iRMX II Operating System.*

RELATED PUBLICATIONS

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

- *iRMX 286 Networking Software User's Guide*, Order Number: 122323
- *iAPX 286 Utilities User's Guide*, Order Number: 121934
- *MULTIBUS® II Transport Protocol Specification,* Order Number: 149247

intel®

CONTENTS

CHAPTER
OVERVIEW PAGE

Introduction 1-1
1.2 Objects 1-2
1.2.1 Tasks 1-2
1.2.2 Jobs 1-3
1.2.3 Segments 1-5
1.2.4 Buffer Pools 1-5
1.2.5.1 Mailboxes 1-5
1.2.5.2 Semaphores 1-5
1.2.5.3 Regions 1-6
1.2.6 Extension And Composite Objects 1-6
1.3 Descriptors 1-6
1.3.1 Descriptor Tables 1-7
1.3.1.1 Global Descriptor Table (GDT) 1-7
1.3.1.2 Local Descriptor Table (LDT) 1-7
1.3.1.3 Interrupt Descriptor Table (IDT) 1-7
1.3.2 Call-Gates 1-7
1.4 Handlers 1-7
1.4.1 Exception Handlers 1-8
1.4.2 Interrupt Handlers 1-8
2.1 Introduction 2-1
2.2 Job Tree and Resource Sharing 2-1
2.3 Job Creation 2-2
2.4 Job Deletion 2-2
2.5 System Calls for Jobs 2-2

CHAPTER 3
TASK MANAGEMENT PAGE

3.1 Introduction 3-1
3.2 Priority 3-1
3.3 Task States 3-1
3.3.1 The Asleep State 3-1
3.3.2 The Suspended State 3-2
3.3.3 The Asleep-Suspended State 3-2
3.3.4 The Ready and Running States 3-2
3.4 Task State Transition 3-2
3.5 Round-Robin Scheduling 3-4
3.6 Additional Task Attributes 3-6
3.7 Task Resources 3-6
3.8 System Calls for Tasks 3-6

CHAPTER 4
EXCHANGE MANAGEMENT PAGE
4.1 Introduction 4-1
4.2 Mailboxes 4-1
4.2.1 Mailbox Queues 4-1
4.2.2 Mailbox Mechanics 4-1
4.2.3 High-Performance Portion of Object Queue 4-2
4.2.4 System Calls for Mailboxes 4-3
4.3 Semaphores4
4.3.1 Semaphore Queue 4-4
4.3.2 Semaphore Mechanics 4-4
4.3.3 System Calls for Semaphores 4-5
4.4 Regions 4-5
4.5 Tasks Involved in Sharing Data 4-6
4.6 Mutual Exclusion Using Semaphores 4-6
4.7 Mutual Exclusion Using Regions 4-7
Usefulness of Semaphores 4-8
4.10 Cautionary Notes on Using Regions 4-10
4.11 System Calls for Regions 4-10

CHAPTER 5
MEMORY MANAGEMENT PAGE
5.1 Introduction 5-1
5.2 Segments 5-1
5.3 Memory Pools 5-2
5.4 Controlling Pool Size 5-2
5.5 Movement of Memory Between Jobs 5-2
5.6 Memory Allocation 5-3
5.6.1 Buffer Pools 5-4
5.7 System Calls for Memory Management 5-4

CHAPTER 6
OBJECT MANAGEMENT PAGE
6.1 Introduction 6-1
6.2 Access Rights 6-1
6.3 Object Address 6-2
6.4 Inquiring About Object Types 6-2
6.5 Using Object Directories 6-2
6.6 System Calls for Objects 6-3

CHAPTER 7
DESCRIPTOR MANAGEMENT PAGE
7.1 Introduction 7-1
7.2 Explicit Descriptors 7-1
7.2.1 Descriptors with Aliases 7-2
7.2.2 Descriptors for Undefined Memory 7-2
7.3 Cautionary Notes on Using Descriptors 7-2
7.4 System Calls for Descriptor Management 7-2

CHAPTER 8
EXCEPTIONAL CONDITION MANAGEMENT PAGE
8.1 Introduction 8-1
8.2 Condition Codes 8-1
8.3 Types of Exceptional Conditions 8-1
8.4 Condition Code Ranges 8-1
8.5 Exception Handlers 8-2
8.6 Assigning an Exception Handler 8-2
8.7 Invoking an Exception Handler 8-3
8.8 Handling Exceptions In-Line 8-4
8.9 Handling Exceptions in 80286 Processor Systems 8-4
8.10 System Calls for Exception Handlers 8-6

CHAPTER 9
INTERRUPT MANAGEMENT PAGE
9.1 Introduction 9-1
9.2 Interrupt Mechanism 9-2
9.2.1 Interrupt Controllers and Interrupt Lines 9-3
9.2.2 Interrupt Levels 9-3
9.2.3 Interrupt Descriptor Table 9-4
9.2.4 Assigning Interrupt Levels to External Sources 9-6
9.2.5 Disabling Interrupts 9-6
Interrupt Handlers and Interrupt Tasks 9-9
9.3.1 Setting Up an Interrupt Handler 9-10
9.3.2 Using an Interrupt Handler 9-11
9.3.3 Using an Interrupt Task 9-11
9.3.3.1 Duties of the Interrupt Handler 9-12
9.3.3.2 Duties of the Interrupt Task 9-12
9.3.3.3 Interrupt Task Priorities 9-12
9.3.4 Interrupt Servicing Patterns 9-15
9.3.5 Using Multiple Buffers to Service Interrupts 9-16
9.3.5.1 Single Buffer Example 9-16
9.3.5.2 Multiple Buffer Example 9-17
9.3.5.3 Specifying The Count Limit 9-19
9.3.5.4 Enabling Interrupt Levels From Within a Task 9-21

9.4 HANDLING SPURIOUS INTERRUPTS 9-22
9.4.1 Calling GET$LEVEL 9-23
9.4.2 Judicious Selection of Interrupt Levels 9-23
9.4.3 Examining the In-Service Register 9-23
9.5 EXAMPLES OF INTERRUPT SERVICING 9-24
9.6 System Calls for Interrupts 9-27

CHAPTER 10
OPERATING SYSTEM EXTENSIONS **PAGE**
10.1 Introduction 10-1
10.2 Three Ways of Adding Functionality 10-1
10.3 Creating an Operating System Extension2
10.3.1 Procedures Used In Operating System Extensions2
10.3.2 Interface Procedures7
10.3.3 Entry Procedures7
10.3.4 Function Procedures8
10.3.5 RQ$ERROR And NUC$ERROR Procedures8
10.4 Establishing Exception-Handling Mechanisms 10-11
10.5 Customized Exception Codes 10-14
10.6 Linking the Procedures 10-14
10.7 Including OS Extensions 10-14
10.8 Protecting Resources From Being Deleted 10-15
10.9 System Calls For Extending The Operating System 10-15

CHAPTER 11
TYPE MANAGERS **PAGE**
11.1 Introduction 11-1
11.2 Creating New Objects 11-1
11.3 Manipulating Composite Objects and Extension Types 11-2
11.4 Deleting Composite Objects and Extension Types 11-3
11.4.1 Type Manager Responsibilities During DELETE$JOB 11-3
11.4.2 Type Manager Responsibilities During DELETE$EXTENSION 11-5
11.4.3 Deletion of Nested Composites 11-5
11.5 Writing a Type Manager 11-6
11.6 Example-A Ring Buffer Manager 11-7
11.6.1 Initialization 11-9
11.6.2 The Interface Library 11-11
11.6.3 The Create Ring Buffer Procedure 11-13
11.6.4 The Delete Ring Buffer Procedure 11-15
11.6.5 The Put Byte Procedure 11-15
11.6.6 The Get Byte Procedure 11-17
11.6.7 Epilogue 11-18
11.7 System Calls for Type Managers 11-18

CHAPTER 12
iRMX® II MULTIBUS® II SYSTEMS

PAGE
12.1 Introduction 12-1
12.2 Features of MULTIBUS® II Systems 12-1
12.3 An Analogy of How MULTIBUS® II Systems Work 12-2
12.4 MULTIBUS® II Hardware Overview 12-3
12.4.1 Central Services Module (CSM) 12-4
12.4.1.1 Global Time-of-Day Clock .... 12-.i.global clock, MULTIBUS II 12-5
12.4.2 Interconnect Address Space .... 12-.i.Interconnect address space 12-5
12.4.3 Built-In Self Tests (BIST) .... 12-.i.Built-in Self Tests (BIST) 12-6
12.4.4 The MULTIBUS® II Message Passing Hardware and Message 12-6
12.5 Extended iRMX® II Software Overview 2-7
12.5.1 The MULTIBUS® II Transport Protocol 12-8
12.5.2 The Nucleus Communication Service 12-8
12.5.3 Nucleus Communication Objects 12-9
12.5.3.1 Port 12-9
12.5.3.2 Buffer Pools 12-10
12.5.3.2.1 System Calls for Buffer Pools 12-11
12.5.3.2.2 Data Chains 12-11
12.5.3.2.3 Message Fragmentation 12-13
12.5.4 System Calls to Work With MULTIBUS® II Message Space 12-13
12.5.4.1 System Calls for Interconnect Space 12-13
12.5.4.2 System Calls for Sending Messages through Message Space 12-13
12.5.4.3 Calls For Getting Information About Message Passing Agents (Boards) 12-14
12.5.5 The Nucleus Communications Service System Calls 12-14
12.5.5.1 System calls used with buffer pool objects 12-14
12.5.5.3 System Calls Used to Send/Receive Messages Through Ports 12-15
12.5.5.4 System Calls used with the Interconnect Registers on a board 12-16
12.5.6 Examples Using Nucleus Communications Service Calls 12-16
12.5.6.1 Interconnect Space Example 12-18
12.5.6.2 Creating a Port for Message Sending and Receiving 12-26
12.5.6.3 Sending Data Using RQ$SEND$RSVP 12-31
12.5.6.4 Sending a Data Chain Message 12-37
12.5.6.5 Sending a Message in Fragments 12-44
12.5.6.6 Receiving a Message in Fragment Form 12-47
12.5.6.7 The Name Server Example 12-54
12.6 Glossary 12-56

CHAPTER 13
NUCLEUS CONFIGURATION

PAGE
12.1 Introduction 12-1
12.2 Hardware 12-1
12.3 System Characteristics 12-2
12.4 System Initialization Error Reporting 12-3

APPENDIX A
EXTENDED iRMX® II DATA TYPES PAGE

APPENDIX B
OBJECT TYPES AND RESOURCE REQUIREMENTS PAGE
B.1 Introduction B-1
B.2 Object Types B-1
B.3 Resource Requirements B-2

APPENDIX C
EXCEPTION CODES PAGE
C.1 Introduction C-1

intel® TABLES

Table 8-1. Condition Code Ranges 8-2
Table 8-2. Return Address after an Exception 8-6
Table 9-1. Allocation of Interrupt Entries 9-5
Table 9-2. Interrupt Levels Disabled for Running Task 9-8
Table 9-3. Interrupt Level and Task Priority Information 9-13
Table 9-4. Handler and Task Interaction through Time 9-20
Table 9-5. Servicing Interrupts with an Interrupt Handler 9-24
Table 9-6. Servicing Interrupts with an Interrupt Task 9-25
Table 9-7. Servicing Interrupts with an Interrupt Handler,
Interrupt Task, and Multiple Buffering 9-26
Table 10-1. Comparison of Techniques for Creating Common Functions 10-2
Table B-1. Type Codes B-1
Table C-1. Exception Codes C-1

intel® FIGURES

Figure 1-1. Initial Job Tree 1-4
Figure 3-1. Task State Transition Diagram 3-4
Figure 5-1. Memory Movement Diagram 5-3
Figure 9-1. Interrupt Processing Mode 9-l2
Figure 9-2. 80286 Interrupt Lines 9-4
Figure 9-3. Flow Chart of Interrupt Handling 9-15
Figure 9-4. Single-Buffer Interrupt Servicing 9-17

FIGURES (continued)

Figure 9-5. Multiple-Buffer Interrupt Servicing 9-18
Figure 10-1. OS Extensions with Entry Procedure 10-4
Figure 10-2. OS Extension without Entry Procedure 10-5
Figure 10-3. Summary of Duties of Procedures in OS Extensions 10-6
Figure 10-4. Handling Exceptions with an Exception Handler 10-10
Figure 10-5. Control Flow for Handling Exceptions In-Line 10-13
Figure 11-1. Creation Sequence for Composite Objects 11-1
Figure 11-2. Type Manager Involvement in DELETE$JOB 11-4
Figure 11-3. A Ring Buffer 11-8
Figure 12-1. Simplified MULTIBUS® II Bus Architecture 12-4
Figure 12-2. A Simplified MULTIBUS® II Message 12-7
Figure 12-3. A Data Chain Block and a Data Chain12-12
Figure 12-4. Physical Location of Boards in the Examples17
Figure 12-5. Board Scanning Algorithm18
Figure 12-6. Implementation of a Board Scanner 12-25
Figure 12-8. Creating a Data Transport Protocol Port 12-30
Figure 12-9. An RSVP/REPLY Transaction between Two Extended iRMX® II Hosts 12-32
Figure 12-10. Algorithm for RQ$SEND$RSVP Example 12-33
Figure 12-11 Algorithm for Server Board 12-33
Figure 12-12. Client Board Code for RQ$SEND$RSVP Example 12-35
Figure 12-13. The Server Board Code to Receive and Answer an RSVP Message 12-37
Figure 12-14. Data Chain Send 12-40
Figure 12-15. Receive a Message in Data Chain Form 12-44
Figure 12-16. Send a Message in Fragments 12-47
Figure 12-17. Receive a Message in Fragments 12-50
Figure 12-18. Sending a Message that Requires Receive Fragmentation 12-52
Figure 12-19. Literal File DCOM.LIT 12-53
Figure 12-20. External File DCOM.EXT 12-54

Intel®

CHAPTER 1 OVERVIEW

INTRODUCTION

The Extended iRMX® II Nucleus is the core of every iRMX II application system. Its activities include

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

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

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

The following generalizations can be made about these types:

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

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

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

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

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

1.2 OBJECTS

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

1.2.1 Tasks

Tasks do the work of the system. They can be considered a virtual CPU. Tasks used by the iRMX II Operating System are software tasks. They should not be confused with 80286 processor hardware tasks. The entire iRMX II Operating System and its tasks, when operating without exceptions, are one hardware task. A task has two goals:

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

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

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

As viewed by the Nucleus, a task is merely a set of values, some of which are

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

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

1. The context of the previously running task is saved by the Nucleus.
2. The Nucleus loads the new running task's context.
3. The new task begins executing.

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

- The task removes itself from the ready state. For example, the task can suspend or delete itself; the task can attempt to receive a token for an object that has not yet been sent, in which case it might elect to wait (in the asleep state).
- The task (task A) is pre-empted when a higher priority task (task B) becomes ready. For example, task B might previously have gone into the asleep state for a specific period of time. When the time period has passed, task B becomes ready again. Because it is then the highest priority ready task, task B becomes the running task.
- The task is rescheduled due to round-robin. For a complete explanation of round-robin scheduling, see Chapter 3.

1.2.2 Jobs

A job consists of tasks and the resources they need.

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

The job tree, as it may look after the initialization of a system, is shown in Figure 1-1.

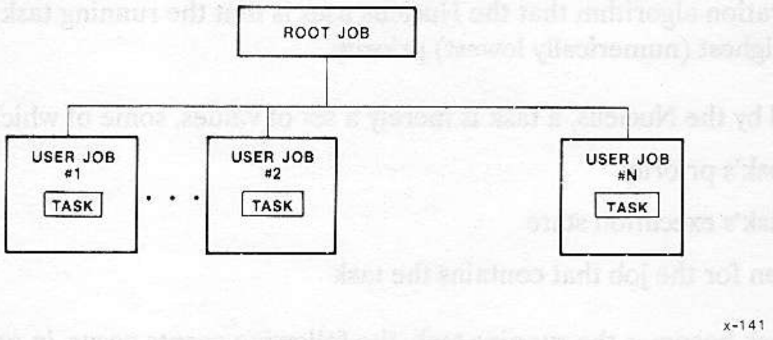


Figure 1-1. Initial Job Tree

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

Also associated with each job is a <u>memory pool</u>. This is an amount of memory, up to 16M bytes, which is allocated to the job and its descendants. All memory needed to create objects in the job comes from the memory pool.

1.2.3 Segments

A fundamental resource that tasks need is memory. Memory is allocated to tasks in the form of segments which are addressable, contiguous blocks of memory containing up to 64K bytes of either code or data. A task needing memory requests a segment of whatever size it requires. The Nucleus attempts to create a segment from the memory pool given to the task's job when the job was created. While the segment is being used by the task, the 80286 microprocessor checks the segment length to ensure that the segment does not read or write beyond the segment length defined. If there is not enough memory available, the Nucleus will try to borrow the needed memory from ancestors of the job. In this respect, the tree-structured hierarchy of jobs is instrumental in resource distribution.

1.2.4 Buffer Pools

Buffer pools are holding areas for segments, you create a buffer pool and then fill it with buffers using the RQ$CREATE$SEGMENT system call. Having a pool of memory readily available cuts down on system overhead because allocating existing buffers is faster than creating and deleting segments.

1.2.5 Exchange Objects

Three of the object types are used as information exchanges: mailboxes, semaphores, and regions. Each of these is explained in the following sections.

1.2.5.1 Mailboxes

A mailbox is one of three types of objects that can be used for intertask communication. When task A wants to send an object or a data packet to task B, task A must send a token for the object or the actual message to a mailbox, and task B must visit that mailbox. If a token or a data packet isn't there, task B has the option of waiting for any desired length of time. If a token is being sent, task B can access the object after obtaining the token. Sending a token for an object in this manner can achieve various purposes. The object might be a segment that contains data needed by the waiting task. On the other hand, the segment might be blank, and sending its token might constitute a signal to the waiting task.

1.2.5.2 Semaphores

A semaphore is a custodian of abstract "units." It dispenses units to tasks that request them, and it accepts units from tasks. Units at a semaphore behave like null messages at a mailbox.

An example of typical semaphore use is mutual exclusion. Suppose your application system contains one I/O device which is being used for output by multiple tasks. To ensure that only one of these tasks can use the device at a given time, you can establish a semaphore which has one unit and require that tasks obtain the unit before using the device. A task wanting to use the device would request the unit from the semaphore. When it gets the unit, it can use the device and then return the unit to the semaphore. Because the semaphore has no units while the task is using the device, other tasks are effectively excluded from using the device. You might want to think of units at a semaphore like currency at a bank. If there is no money in the bank, the bank cannot function.

1.2.5.3 Regions

A region is a one-unit semaphore with special semantics. It is an iRMX II object that tasks can use to restrict access to a specific collection of shared data. Once a task gains access to shared data through a region, by issuing a successful ACCEPT$CONTROL system call, the task can not be suspended or deleted (although it may still be pre-empted by a higher priority task) by other tasks until it surrenders access. When the task currently using the shared data no longer needs access, it notifies the operating system, which then allows the next task to access the shared data.

1.2.6 Extension And Composite Objects

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

1.3 DESCRIPTORS

The Nucleus keeps track of each object by means of a 16-bit value called a token. The token contains the logical address of the object. However, a descriptor is needed to determine the physical address. Descriptors are used to give an area of memory addressability. Each descriptor is an entry in a descriptor table and contains the physical address of a segment. The operating system assigns each object a descriptor when it is created. Every object must have at least one descriptor or there is no way to address it.

1.3.1 Descriptor Tables

All descriptors reside in a hardware descriptor table. There are three types of descriptor tables: the Global Descriptor Table (GDT), the Local Descriptor Table (LDT), and the Interrupt Descriptor Table (IDT).

1.3.1.1 Global Descriptor Table (GDT)

The GDT is a table of up to 8K entries each of which is a descriptor containing the 24-bit physical address used by the system to access areas of memory. Descriptors in the GDT can be used by every task in the system. There is only one GDT for the entire operating system. All the descriptors you need will be in the GDT.

1.3.1.2 Local Descriptor Table (LDT)

The LDT is the only hardware LDT in the iRMX II Operating System. It is reserved for system use. Additional LDTs are not available.

1.3.1.3 Interrupt Descriptor Table (IDT)

The IDT is a table containing the address of the interrupt handling code to be executed when an interrupt occurs. Addresses can be entered into the IDT either when the system is created or dynamically using the SET$INTERRUPT system call.

1.3.2 Call-Gates

Call-gates are used to enter the iRMX II Operating System and OS extensions. They redirect flow within a task from one code segment to another. Each system call uses a call-gate to transfer the program directly to the iRMX service routine requested. Call-gates are part of the descriptor tables and are reserved when the system is configured.

1.4 HANDLERS

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

1.4.1 Exception Handlers

Tasks occasionally make errors. If an error occurs during an iRMX II system call, it causes an exceptional condition. If an error occurs as a result of a hardware protection feature, such as, a program trying to execute out of its segment bounds or trying to execute a segment that is defined as read only, it causes an exceptional condition known as a trap. The occurrence of an exceptional condition or a trap can, if desired, cause a transfer of control to the exception handler associated with the current task. The exception handler is a procedure that typically deals with the problem by one of the following methods:

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

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

1.4.2 Interrupt Handlers

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

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

CHAPTER 2
JOB MANAGEMENT

2.1 INTRODUCTION

A job is an environment in which iRMX II objects such as tasks, mailboxes, semaphores, segments, and (offspring) jobs reside. In addition, a job has an object directory and a memory pool of up to 16M bytes. The job's memory pool provides the raw material from which objects can be created by the tasks in the job.

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

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

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

2.2 JOB TREE AND RESOURCE SHARING

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

Associated with each job is the following set of limits:

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

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

For example, if job A creates job B, these events occur:

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

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

2.3 JOB CREATION

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

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

2.4 JOB DELETION

Before a job can be deleted, all of its extension objects (see Chapter 11) and descendant jobs must be deleted. By using the RQE$OFFSPRING system call, the deleting task can probe down the job tree and find all of the descendants. Then it can delete them, beginning with descendants that have no children and working up the tree. After all of the descendants have been deleted, the task can delete the target job.

2.5 SYSTEM CALLS FOR JOBS

The following system calls manipulate jobs:

- RQE$CREATE$JOB--creates a job with a memory pool of up to 16M bytes and returns a token for the job; resources for the new job are drawn from the resources of the job to which the invoking task belongs. This system call should be used for all new applications or for present applications which may expand beyond 1M byte.

- CREATE$JOB--creates a job with a memory pool of up to 1M byte and returns a token for the job; resources for the new job are drawn from the resources of the job to which the invoking task belongs. This call is available for compatibility with the iRMX 86 Operating System. It should be used only by applications that require compatibility with the iRMX 86 Operating System. All new applications should use RQE$CREATE$JOB.
- DELETE$JOB--deletes a childless job that contains no extension objects and returns the job's resources to its parent.
- RQE$OFFSPRING--provides a list of the child jobs of the specified job in a user-supplied data structure.
- OFFSPRING--provides a segment containing tokens of the child jobs of the specified job.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

CHAPTER 3 TASK MANAGEMENT

3.1 INTRODUCTION

Tasks are the active objects in an iRMX II system. Each task is part of a job and is restricted to the resources that its job provides.

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

3.2 PRIORITY

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

Unless a task is involved in processing interrupts (see Chapter 9), its priority should be between 128 and 255. When a task having a priority in the range 0 to 127 is running, certain external interrupt levels are disabled, depending on the priority.

3.3 TASK STATES

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

3.3.1 The Asleep State

A task is in the asleep state when it is waiting for a request to be granted. A task can put itself in the asleep state by issuing a SLEEP system call, or it can be placed there by the operating system after issuing a request that cannot be granted immediately if the task is willing to wait. In either case, the length of time the task stays in the asleep state is controlled by a parameter that the task specifies.

3.3.2 The Suspended State

A task enters the suspended state when it is placed there by another task, when it is waiting for an interrupt, or when it suspends itself. Associated with each task is a suspension depth, which reflects the number of "suspends" outstanding against it. Each suspend operation must be countered with a resume operation before the task can leave the suspended state. The suspension level (the number of outstanding suspends) is not available as a service of the operating system.

3.3.3 The Asleep-Suspended State

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

3.3.4 The Ready and Running States

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

3.4 TASK STATE TRANSITION

The Nucleus allocates processor time to tasks in a priority-based manner. The following discussion explains this method of allocating processor time.

As an iRMX II application system runs, events occur which cause tasks to pass from state to state. The iRMX II Operating System is, therefore, event-driven. Figure 3-1 shows the paths of transition between states.

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

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

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

- The task has just become ready and has higher priority than any other ready task.
- The task is ready and no other task of equal or higher priority is before it in the ready queue (see section, "Round-Robin Scheduling," for further explanation).

(3) The task goes from the running state to the ready state when the task is pre-empted by a higher priority task that has just become ready or when the task is rescheduled as a result of round-robin (see section "Round-Robin Scheduling").

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

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

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

- The time period specified in the invocation of the SLEEP system call expires.
- The task's designated waiting period expires without its request being granted.
- The task's request is granted (because another task issued a system call such as SEND$MESSAGE or SEND$UNITS that sends a message, and the message was received).
- The object at which the task was waiting was deleted (for example, mailbox).

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

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

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

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

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

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

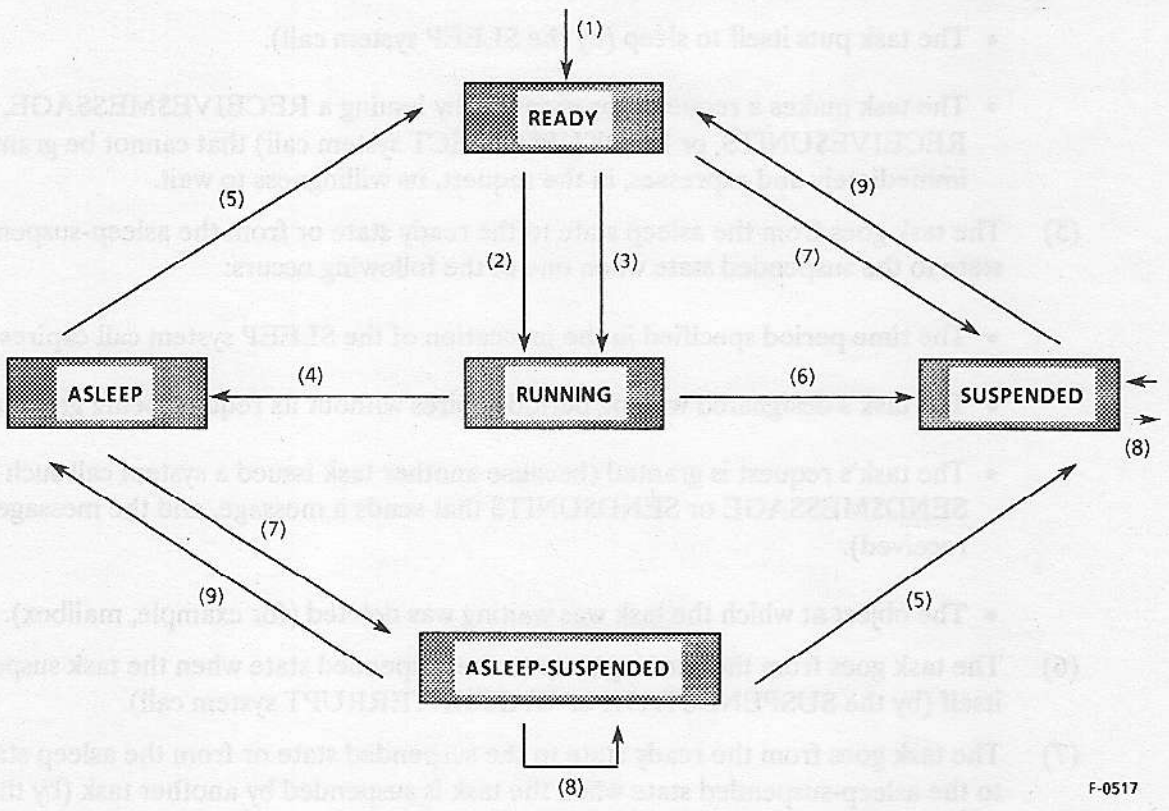


Figure 3-1. Task State Transition Diagram

3.5 ROUND-ROBIN SCHEDULING

As mentioned previously, the iRMX II Operating System schedules tasks based on priority. Two tasks with the same priority compete for CPU resources, and often one task is left waiting indefinitely. To prevent this from happening, the iRMX II Operating System offers round-robin scheduling.

Round-robin scheduling is particularly desirable in a multi-user development environment. It prevents one task from monopolizing the CPU while other tasks wait indefinitely.

There are a number of factors which can cause a task to lose control of the CPU.

- The task is pre-empted by a hardware interrupt.

- The task is pre-empted by a higher priority task.
- The task performs a system call that causes it to relinquish control of the CPU.

In the first two cases, the task (referred to throughout the text as task A) remains in the READY state. Without round-robin scheduling, when the higher priority activity is completed, task A regains control. Thus, any other tasks having the same priority as task A will not run until task A performs a system call causing it to relinquish control of the CPU.

With round-robin scheduling, task A is allocated a time quota. When its time quota expires, it is pre-empted. If there are other tasks of the same priority in the READY state, task A loses control of the CPU to another task, and is placed in the ready list after all tasks of the same priority. Task A regains control only when it reaches the top of the list.

Round-robin scheduling affects only tasks that are of lower priority (numerically higher) than a level you determine when configuring the system. The recommended threshold priority level of 140 ensures that high-priority, time-critical tasks such as interrupt tasks are not affected. The Nucleus always executes the highest priority task until it is pre-empted by a higher priority ready task or until it relinquishes control.

The following example illustrates the advantage of using round-robin scheduling. Assume you have tasks A, B, and C. Task C has priority 130, and tasks A and B have priority 200. Round-robin scheduling has been configured with a time quota of 5 ticks and a threshold priority of 140. Task A runs for 2 clock ticks when task C becomes ready. Task C immediately gains control because of its higher priority. When task C relinquishes control of the CPU, task A continues to run for its remaining 3 clock ticks. It is then pre-empted and task B runs. After task B relinquishes control or is completed, task A is rescheduled for another 5 clock ticks. Without round-robin scheduling, task B would not run as task A would continue running until a higher priority task became ready or it relinquished the processor.

To implement round-robin scheduling, it is necessary to configure two parameters on the "Nucleus" screen of the Interactive Configuration Utility. These parameters establish the threshold priority level and the time quota each task can run before it is pre-empted. The default threshold priority is 255, which means round-robin is turned off. To use round-robin, the recommended threshold priority is 140. The default time quota is 50 milliseconds. For more details on configuring the Nucleus, see the *iRMX II Interactive Configuration Utility Reference Manual.*

3.6 ADDITIONAL TASK ATTRIBUTES

In addition to priority, execution state, and suspension depth, the Nucleus maintains current values of the following attributes for each existing task: containing job, its register context, starting address of its exception handler (see Chapter 8), its exception mode (see Chapter 8), whether or not it is an interrupt task (see Chapter 9) and whether the task uses the Numeric Extension Processor (NPX).

3.7 TASK RESOURCES

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

3.8 SYSTEM CALLS FOR TASKS

The following system calls are provided for task manipulation:

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

For a complete list and explanation of the iRMX II Nucleus system calls, see the *iRMX II Nucleus System Calls Reference Manual.*

Intel®

CHAPTER 4 EXCHANGE MANAGEMENT

4.1 INTRODUCTION

The iRMX II Nucleus provides exchanges to facilitate intertask communication, synchronization, and mutual exclusion. When a task uses an exchange, it is always acting either as a sender or as a receiver. There are three kinds of exchanges: mailboxes, semaphores, and regions. If the exchange is a mailbox, one task will send a message to the mailbox; another task will go to the mailbox to receive the object's message. If the exchange is a semaphore, either a task is receiving units from the semaphore, or it is sending units to the semaphore. If the exchange is a region, the data in the region can be accessed by only one task at a time, and this task cannot be deleted or suspended until it relinquishes control.

4.2 MAILBOXES

Mailboxes support intertask communication. A sending task uses a mailbox to pass either a token or a message of up to 128 bytes to another task. For example, the object might be that of a segment containing data needed by the receiving task.

4.2.1 Mailbox Queues

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

4.2.2 Mailbox Mechanics

When creating a mailbox, you must specify whether the mailbox is a data mailbox or a message mailbox. Data mailboxes are manipulated with the system calls SEND$DATA and RECEIVE$DATA, whereas, message mailboxes are manipulated with the SEND$MESSAGE and RECEIVE$MESSAGE system calls. If you try passing a message to a mailbox with the wrong system call, for example sending a token with SEND$DATA, the Nucleus issues an E$TYPE exception code.

When a task sends either a token or a data packet to mailbox, using the SEND$MESSAGE or the SEND$DATA system call, one of two events occurs. If no tasks are waiting at the mailbox, the message is placed at the rear of the message queue (which might be empty). Message queues (object or data queues) are processed in a first-in/first-out (FIFO) manner, so the message remains in the queue until it moves to the front and is given to a task.

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

NOTE

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

When a task attempts to receive a message from a mailbox via the RECEIVE$MESSAGE or RECEIVE$DATA system call, and the message queue at the mailbox is not empty, the task receives the message immediately and remains ready. However, if there are no messages at the mailbox one of two events occurs:

- If the task specifies in the time$limit parameter that it is willing to wait, it is placed in the mailbox's task queue and is put to sleep. If the designated waiting period elapses before the task gets a message, the task is made ready and receives an E$TIME exceptional condition (see Appendix D for a list of error conditions).
- If the task specifies in the time$limit parameter that it is not willing to wait, it remains ready and receives an E$TIME exceptional condition.

A task has the option, when using the SEND$MESSAGE system call, of specifying that it wants acknowledgment from the receiving task. Thus, any task using the RECEIVE$MESSAGE system call should check to see if an acknowledgment has been requested. This option is not available to a task using the SEND$DATA system call. For details, see the description of the system calls in the *Extended iRMX II Nucleus System Calls Reference Manual*.

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

4.2.3 High-Performance Portion of Object Queue

The object queue of each mailbox is divided into two portions: a high-performance portion and an overflow portion. The high-performance portion is directly associated with each mailbox, while the overflow portion is created by the operating system as needed.

A task, when creating a mailbox with CREATE$MAILBOX, can specify the number of objects the high-performance portion can hold, from 4 to 60. By using this high-performance portion of the object message queue, the task can greatly improve the performance of SEND$MESSAGE and RECEIVE$MESSAGE when these calls actually get or place objects on the queue (the high-performance portion has no effect when tasks are already waiting at the task queue). When more objects are queued at a mailbox than the high-performance portion can hold, the objects overflow into an extra buffer that holds up to 100 messages. The overflow buffer is not deleted until the object message queue empties. Thus, the average slowdown experienced when the high-performance portion overflows is almost negligible.

The high-performance portion has high speed because the Nucleus allocates memory for it while creating the mailbox. The Nucleus allocates this memory permanently to the mailbox, even if no objects are queued there. No space is allocated for the overflow portion of the queue until the space is needed to contain objects. However, because an overflow buffer is not created for every send/receive message, but rather for every 100 messages, there is almost no effect on performance. Performance is affected only in the worst-case when a SEND$MESSAGE system call causes the allocation of an overflow buffer. In this case, extra time is required for the allocation. If you know the number of objects you will have, it is advisable to configure the high-performance portion to hold them.

4.2.4 System Calls for Mailboxes

The following system calls manipulate mailboxes:

- CREATE$MAILBOX--creates a mailbox and returns a token.
- DELETE$MAILBOX--deletes a mailbox from the system.
- SEND$DATA--sends a data packet of up to 128 bytes to a mailbox.
- SEND$MESSAGE--sends an object to a mailbox.
- RECEIVE$DATA--receives a data packet from a mailbox; the task has the option of waiting if no data packets are present.
- RECEIVE$MESSAGE--receives an object from a mailbox; the task has the option of waiting if no objects are present.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

4.3 SEMAPHORES

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

4.3.1 Semaphore Queue

Semaphores have only one queue, a task queue. The task queue is either FIFO or priority-based. The queueing scheme to be used is specified for each semaphore at the time of its creation.

4.3.2 Semaphore Mechanics

Because tasks can request more than one unit, a semaphore might simultaneously have both tasks in its queue and units in its custody. That scheme is best understood by imagining that the semaphore is trying, at all times, to satisfy the request of the task which is at the front of the semaphore's task queue. Only when it can provide as many units as the task requested does it award units, and then it does so immediately. A request made to a semaphore is either granted in full or it is not granted at all.

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

When a task requests units from a semaphore via the RECEIVE$UNITS system call, the request must be within the specified maximum for that semaphore; otherwise, the request is invalid and causes an E$LIMIT exceptional condition. If a task's request for units is valid and if the size of the request is within the semaphore's current supply of units and, the task is at the front of the semaphore's task queue (or would be if queued), then the request is granted immediately and the task remains ready. Otherwise, one of the following applies:

- If the task specifies in its time$limit parameter that it is willing to wait, it is placed in the semaphore's task queue and is put to sleep. If the designated waiting period elapses before the task gets its requested units, the task is made ready and receives an E$TIME exceptional condition.
- If the task specifies in its time$limit parameter that it is not willing to wait, it remains ready and receives an E$TIME exceptional condition.

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

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

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

NOTE

A task sending units to a semaphore can be pre-empted by a higher priority task becoming ready as a result of receiving its requested units.

4.3.3 System Calls for Semaphores

The following system calls manipulate semaphores:

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

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

4.4 REGIONS

A region is an iRMX II 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.

Regions should be restricted to specific uses. Misuse of regions can have profound affects on your application system.

4.5 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 task 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. While task A fetches the X-coordinate of the position it is pre-empted by task B before it can fetch 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. (That is, one of the tasks could possibly pre-empt 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 pre-empted by other tasks desiring access. This protocol for sharing data is called <u>mutual exclusion</u>.

4.6 MUTUAL EXCLUSION USING SEMAPHORES

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, 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 pre-empts task A (low priority).

In Step 2, task C must wait for task A (which has lower priority) to finish using the shared data, since 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.

4.7 MUTUAL EXCLUSION USING REGIONS

Tasks can use regions as well as semaphores to obtain mutual exclusion. However, you should note these facts about 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 (although it many still be pre-empted) 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.

You should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be stopped asynchronously (via CONTROL-C entered at a terminal) while the task is accessing data guarded by the region.

- When you create a region you must specify which of two rules (FIFO or priority) is to be used to determine which waiting task next gains access to the shared data.

4.8 USEFULNESS OF SEMAPHORES

Despite the seeming drawbacks of semaphores, there are three reasons to use them:

1. You can use semaphores to accomplish much more than mutual exclusion. For example, with semaphores you can synchronize multiple tasks or allocate resources. Regions, on the other hand, provide only mutual exclusion.
2. Because of the possibility of deadlock, regions should not be used outside of extensions to the operating system. Consequently, programmers not familiar with operating system extensions must use semaphores to accomplish mutual exclusion.
3. Semaphores allow a task to set an upper limit on the amount of time the task is willing to wait for access. In contrast, regions provide no such option. Tasks using regions for mutual exclusion have only two choices:
   - They can request immediate access (with the ACCEPT$CONTROL system call). If a task makes such a request and access is not available immediately, the task does not wait at the region. Rather, it receives an exception code and continues to run.
   - They can request access as it becomes available (with the RECEIVE$CONTROL system call). 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.

4.9 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 how deadlock can occur in using nested regions and how to avoid this situation.

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, called 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 pre-empts 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 adhering 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 and relinquishes the regions in **reverse** 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.

4.10 CAUTIONARY NOTES ON USING REGIONS

Use of regions should be restricted to programmers that have a firm understanding of the operating system and the entire application system. A less-knowledgeable 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, a programmer can make tasks immune to deletion. If they never surrender access, the tasks can permanently avoid deletion.

Abusing some of the features described in this manual (such as regions) can affect the integrity of the entire operating system. If you wish to preserve the integrity of your application system, confine the use of regions to programmers writing operating system extensions.

4.11 SYSTEM CALLS FOR REGIONS

The following system calls manipulate regions:

- ACCEPT$CONTROL--allows a task to gain access to shared data only when access is immediately available. If a different task already has access, the requesting task remains ready but receives an exception code.
- CREATE$REGION--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$REGION--deletes a region.
- RECEIVE$CONTROL--causes a task to wait at the region until the task gains access to the shared data.
- SEND$CONTROL--when issued by a task, frees the operating system to grant a different task access to the shared data.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

Intel®

CHAPTER 5 MEMORY MANAGEMENT

5.1 INTRODUCTION

Occasionally a task needs additional memory. The Free Space Manager in the Nucleus supplies the run time memory required by the jobs in an Extended iRMX II Operating System. All free space initially belongs to the root job. By using Nucleus system calls for allocating and deallocating memory, tasks can usually satisfy their memory needs.

5.2 SEGMENTS

Allocated memory is treated as a collection of segments. A segment is a contiguous sequence of memory not exceeding 64K bytes. A segment's physical starting address is on an even byte boundary (that is, it is divisible by 2). A segment is assigned a slot (descriptor) in the GDT. That GDT slot serves as the segment token. You can access a segment by loading the specific slot number into a selector.

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

NOTE

The token for a segment can be used as the selector of a pointer to the segment. Thus, the token can be used as a selector (as when writing a message in the segment) or as an object reference (as when sending the segment-with-message to a mailbox). The PL/M-286 SELECTOR data type is especially useful in referring to the segment.

5.3 MEMORY POOLS

A memory pool is the memory available to a job and its descendants. Each job has a memory pool. When a job is created, the memory for its pool is allocated from the pool of its parent job. Thus, there is a tree-structured hierarchy of memory pools, identical in structure to the hierarchy of jobs. Memory that a job borrows from its parent remains in the pool of the parent as well as being in the pool of the child. Such memory, however, is only available to tasks in the child job, and not to tasks in the parent job, until the child job releases the borrowed memory.

5.4 CONTROLLING POOL SIZE

Two parameters, pool$min and pool$max, of the RQE$CREATE$JOB system call (and CREATE$JOB, although it has been retained for compatibility only), dictate the range of sizes of a new job's memory pool. The job's memory pool can be up to 16M bytes. Initially, the memory pool is physically contiguous and is equal to pool$min, the pool minimum. If the task needs more memory, it may borrow the memory from its parent job. In this case, the memory requested is a contiguous memory block, but is is not contiguous to the initial memory pool. The maximum amount of memory that may be borrowed is equal to

pool$max - pool$min

Memory allocated to tasks in the child job is still considered to be in the job's pool. A task needing to know about its job's pool or another job's pool may use the RQE$GET$POOL$ATTRIB system call to obtain pool$min, pool$max, the initial pool size, the number of paragraphs currently available, the number of paragraphs currently allocated, and the amount of memory borrowed.

5.5 MOVEMENT OF MEMORY BETWEEN JOBS

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

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

5.6 MEMORY ALLOCATION

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

When a task requests memory, the memory is allocated in segments 18 bytes longer than the specified size. These 18 bytes are for internal use by the Nucleus. However, each selector returned points to the first address available to the task.

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

When a task no longer needs a segment, it can return the segment to the unallocated part of the job's pool by using the DELETE$SEGMENT system call. Because of the algorithm used by the Free Space Manager for returning segments to the memory pool, memory fragmentation is minimal and has little effect on performance. Figure 5-1 shows how memory "moves."

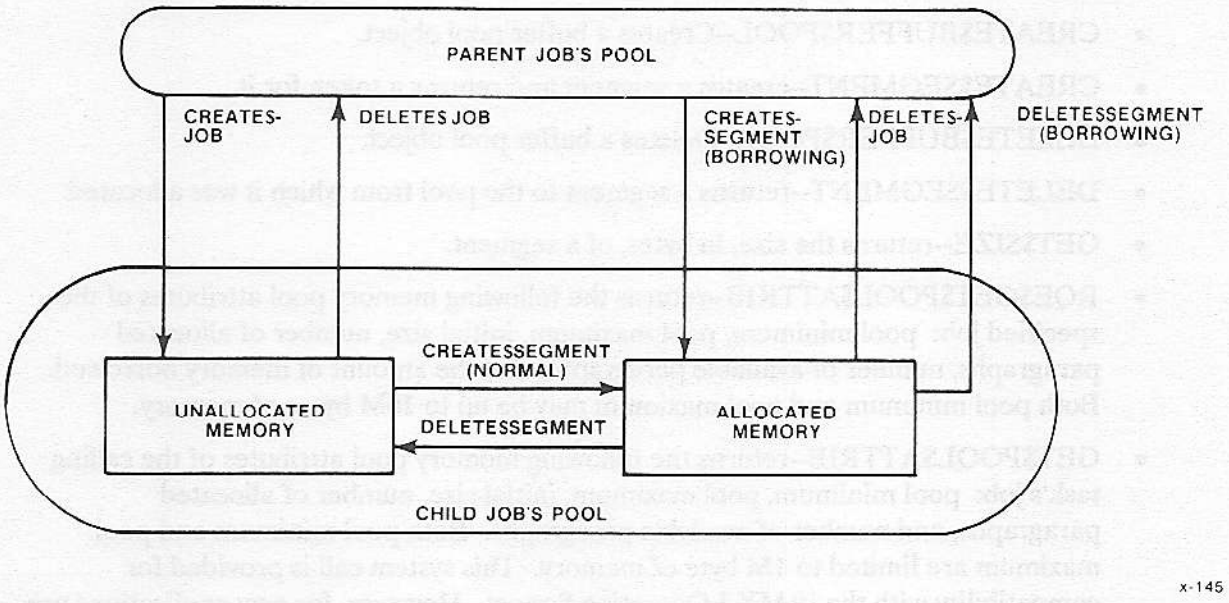


Figure 5-1. Memory Movement Diagram

5.6.1 Buffer Pools

Buffer Pools are holding areas for segments which are used by tasks when needed. Having a pool of memory readily available to tasks cuts down on system overhead because allocating the existing buffers is faster than creating and deleting segments.

Buffer pools are empty when created. The user gives segments to the buffer pool. The segments are created using the the RQ$CREATE$SEGMENT system call. The created segments are given to a buffer pool by using the RQ$RELEASE$BUFFER system call. The buffers are then used by tasks that require memory. Any task that requires frequent creation and deletion of segments may improve performance by using a buffer pool with pre-allocated segments.

Buffer pools incur a certain amount of system overhead in their creation. The following formula defines the amount of resources required.

(Max Buffers * 4) + 108 bytes = the amount of memory used by any given buffer pool.

When you create a buffer pool you specify the following information:

- The maximum number of buffers that can reside in the buffer pool at any one time (8192 maximum.)

5.7 SYSTEM CALLS FOR MEMORY MANAGEMENT

The system calls for memory management are

- CREATE$BUFFER$POOL--Creates a buffer pool object.
- CREATE$SEGMENT--creates a segment and returns a token for it.
- DELETE$BUFFER$POOL--Deletes a buffer pool object.
- DELETE$SEGMENT--returns a segment to the pool from which it was allocated.
- GET$SIZE--returns the size, in bytes, of a segment.
- RQE$GET$POOL$ATTRIB--returns the following memory pool attributes of the specified job: pool minimum, pool maximum, initial size, number of allocated paragraphs, number of available paragraphs, and the amount of memory borrowed. Both pool minimum and pool maximum may be up to 16M bytes of memory.
- GET$POOL$ATTRIB--returns the following memory pool attributes of the calling task's job: pool minimum, pool maximum, initial size, number of allocated paragraphs, and number of available paragraphs. Both pool minimum and pool maximum are limited to 1M byte of memory. This system call is provided for compatibility with the iRMX I Operating System. However, for new applications use RQE$GET$POOL$ATTRIB.
- RELEASE$BUFFER--Return a (segment) buffer to a previously created buffer pool.

- REQUEST$BUFFER--Get a buffer (segment) from a buffer pool that has been supplied with buffers via the RQ$CREATE$SEGMENT system call.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

CHAPTER 6 OBJECT MANAGEMENT

6.1 INTRODUCTION

The Nucleus provides the objects from which the other subsystems are constructed. The Extended iRMX II Nucleus uses 16-bit values, called tokens, to manage the objects in the system. Tokens, as defined by the iRMX II Operating System, are logical addresses. They are selectors that reference an entry in the global descriptor table (GDT). It is the GDT entry which contains the 24-bit physical address of the memory used by the object.

6.2 ACCESS RIGHTS

One of the protection features of the 80286 processor is the access byte. This byte contains the attributes of an 80286 segment, that is, it defines the way a segment can be used by instructions in other 80286 segments. When an iRMX II object is created, its corresponding segment is assigned a read/write access type. Before any operation is performed, the hardware checks the access type. If you have entered the wrong access type, the hardware causes an exception. The iRMX II Operating System has taken advantage of this hardware feature by allowing a task to change an object's access type for segment objects, descriptor objects or composite objects. Access rights for all other object types (job, task, mailbox, semaphore, region and extension) cannot be changed.

Two system calls are provided for manipulating the access byte. RQE$GET$OBJECT$ACCESS supplies the value of the object's access byte. The RQE$CHANGE$OBJECT$ACCESS system call allows you to change an object's access rights. It uses the access byte format provided by the 80286 processor for both code and data segment descriptors. For a list of the possible access byte values, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

NOTE

Do not try to change bits in a token. This may cause a hardware trap.

6.3 OBJECT ADDRESS

The RQE$GET$ADDRESS system call converts an object's logical address into its 24-bit physical address. The physical address may be necessary when using device drivers or when creating aliases as part of descriptor management (see Chapter 7).

6.4 INQUIRING ABOUT OBJECT TYPES

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

6.5 USING OBJECT DIRECTORIES

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

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

NOTE

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

If the object has not yet been cataloged, and the task is not willing to wait, the task remains ready and receives an E$TIME exceptional condition (unless the object directory is full, in which case the task receives an E$LIMIT condition code). However, if the task is willing to wait, it is put to sleep; then two possibilities exist:

- If the designated waiting period elapses before the task receives its requested token, the task is made ready and receives an E$TIME exceptional condition (see Appendix D).
- If the task receives its requested token within the designated waiting period, it is made ready with no exceptional condition. This case is possible because another task can catalog the appropriate entry in the specified object directory while the requesting task is waiting.

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

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

6.6 SYSTEM CALLS FOR OBJECTS

The following system calls manipulate objects:

- CATALOG$OBJECT--places an object in an object directory.
- GET$TYPE--accepts a token for an object and returns its type code.
- LOOKUP$OBJECT--accepts a cataloged name of an object and returns a token for it.
- RQE$CHANGE$OBJECT$ACCESS--changes the access byte of an object.
- RQE$GET$ADDRESS--returns the physical address of an object.
- RQE$GET$OBJECT$ACCESS--returns the value of an object's access byte.
- UNCATALOG$OBJECT--removes an object from an object directory.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual*.

Intel®

CHAPTER 7 DESCRIPTOR MANAGEMENT

7.1 INTRODUCTION

Descriptors are used to address an area of memory. Every segment must have at least one descriptor or it is not addressable. Each descriptor is an entry in the GDT and contains the physical address, the access rights, and the segment limits.

The Nucleus assigns each object a descriptor when it is created. This type of descriptor may be thought of as an implicit descriptor. Implicit descriptors are managed by the Operating System. Application programmers do not need any additional information about descriptors and may want to skip the rest of this section.

7.2 EXPLICIT DESCRIPTORS

The system programmer should know that there is a second type of descriptor which can be thought of as an explicit descriptor. Explicit descriptors are used primarily for the following purposes:

- To gain addressability to areas of memory that are not defined when the system is configured and thus, have no logical address.
- To create aliases to existing segments. (Aliases are one of several descriptors that may be necessary to define a different segment type or a different access right for the same segment.)
- To add device drivers to the system. (See the *Extended iRMX II Basic I/O System User's Guide*.)

You can manipulate descriptors like segments. You can create, change and delete them. In fact, to the operating system they look just like segments. If you call GET$TYPE on a descriptor, the type code returned is for a segment.

Great care should be taken when creating a descriptor. By calling RQE$CREATE$DESCRIPTOR it is possible to create a descriptor for any physical address. An error in calculating the physical address may overwrite valuable system information such as the GDT. When you create a descriptor a hardware slot is established in the GDT with the required physical address. The Nucleus marks the object as a descriptor. In this way, the Nucleus "knows" that when the descriptor is deleted, using RQE$DELETE$DESCRIPTOR, only the GDT slot is to be recycled, not the memory addressed by the descriptor.

The iRMX II Operating System also provides the RQE$CHANGE$DESCRIPTOR system call which can be used to change the physical address of a descriptor and/or the length of the segment addressed. This system call is particularly useful in applications that include I/O drivers.

7.2.1 Descriptors with Aliases

As stated above, you can use descriptors with aliases and with areas of memory that were not defined at configuration time. Aliases allow you to have several descriptors for the same segment. They provide segments with alternate names in much the same way as people use nicknames. Deleting an alias descriptor does not delete the segment to which it refers.

7.2.2 Descriptors for Undefined Memory

Descriptors can also be used to gain addressability to areas of memory that were not defined when the system was configured and thus, have no logical address. These memory areas are not allocated from the job's memory pool. When they are created they do not reduce the size of the memory pool. Therefore, when they are deleted, they do not return memory to the memory pool.

7.3 CAUTIONARY NOTES ON USING DESCRIPTORS

Descriptors are a very powerful feature of the operating system. If they are misused, they can can affect the integrity of the entire operating system. If you wish to preserve the integrity of your application system, confine the use of descriptors to experienced programmers who have a firm understanding of iRMX II addressing. A less-knowledgeable programmer can, by abusing descriptors, corrupt the interaction between tasks in an application system.

7.4 SYSTEM CALLS FOR DESCRIPTOR MANAGEMENT

The following system calls manipulate descriptors.

- RQE$CREATE$DESCRIPTOR--returns a segment token for an entry in the GDT.
- RQE$CHANGE$DESCRIPTOR--changes the physical address contained in the GDT and/or the size of the segment described.
- RQE$DELETE$DESCRIPTOR--returns a slot from the GDT to the operating system for reuse.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

Intel®

CHAPTER 8 EXCEPTIONAL CONDITION MANAGEMENT

8.1 INTRODUCTION

When a task invokes an iRMX II system call, sometimes the results are not what the task is trying to achieve. For example, a segment access may overflow its boundaries or a task may request memory that is not available. In such cases, the operating system must inform the task that an error occurred. Whenever a task makes a system call or the 80286 processor traps an illegal condition, the system uses a condition code to communicate the success or failure of the call.

8.2 CONDITION CODE VALUES AND MNEMONICS

Condition codes are numeric values that represent unique conditions. Each condition code also has a mnemonic (such as E$OK), to indicate the code's meaning. Appendix D lists the condition codes with their numeric values and mnemonics.

When writing application tasks, you can refer to the condition codes by their mnemonics as long as you declare each mnemonic and its numeric code to be literally equal. Intel supplies the /RMX286/INC/ERROR.LIT file which contains literal declarations of all the iRMX II condition code mnemonics.

8.3 TYPES OF EXCEPTIONAL CONDITIONS

Conditions that represent failure (or not complete success) are called exceptional conditions. These conditions have two classifications: programmer errors and environmental conditions. A programmer error is a condition that the calling task can prevent. In contrast, an environmental condition arises outside the control of the calling task. See Appendix D for a complete list of both programmer errors and environmental conditions.

8.4 CONDITION CODE RANGES

The values of condition codes fall into ranges based on the iRMX II layer which first detects the condition. Table 8-1 lists the layers and their respective ranges, with numeric values expressed in hexadecimal notation.

Table 8-1. Condition Code Ranges

Layer	Environmental Conditions	Programming Errors
Nucleus	0H to 0FH	8000H to 800FH
I/O Systems	20H to 5FH	8020H to 805FH
Application Loader	60H to 7FH	8060H to 807FH
Human Interface	80H to AFH	8080H to 80AFH
Universal Development Interface	C0H to DFH	80C0H to 80DFH
Reserved for Intel	E0H to 3FFFH	80E0H to BFFFH
Reserved for users	4000H to 7FFFH	C000H to FFFFH

8.5 EXCEPTION HANDLERS

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

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

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

- The condition code (WORD).
- A code (BYTE) indicating which parameter, if any, was faulty in the call (1 for first, 2 for second, etc., 0 if none).
- A reserved (WORD) parameter.
- A (WORD) parameter containing the Numeric Processor Extension (NPX) status word. This parameter is valid only if the condition code is E$NDP$ERROR.

8.6 ASSIGNING AN EXCEPTION HANDLER

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

A system exception handler is provided as part of the iRMX II Operating System. When you configure the system, you may specify the System Debugger, SDB, as the system exception handler (this is convenient for debugging). In this case, the iRMX II Operating System lets the monitor and the SDB debugger take control of all the 80286 hardware exceptions (except those that handle the Numeric Processor Extension). This means that the monitor, in conjunction with the SDB debugger, will always handle hardware exceptions (causing a break to the monitor, and sending a message to the console), even for iRMX II tasks that specify their own exception handler. A user-written exception handler may still be invoked to handle errors detected in the iRMX II system calls.

If you want to write your own exception handlers, compile them using the PL/M-286 LARGE control, specifying the PUBLIC attribute. It is also possible to compile exception handlers using the COMPACT control, as long as the following conditions are met:

- One extra dummy word parameter is added to the calling sequence (at the end of the parameter list).
- The exception handler must be in the same code segment as the task it serves.
- The exception handler does not handle hardware traps.

8.7 INVOKING AN EXCEPTION HANDLER

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

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

When the Nucleus detects that a task has caused an exceptional condition in making a system call, it compares the type of the condition with the calling task's exception mode. If a transfer of control is indicated, the Nucleus passes control to the exception handler on behalf of the task. The exception handler then deals with the problem, after which control returns to the task, unless the exception handler deleted the task. When the exception handler returns, the task can also detect that an error occurred, because the system call's except$ptr parameter points to a word containing the condition code. While the exception handler is executing, the errant task is still regarded by the Nucleus to be the running task. Therefore, the exception handler task uses the stack and environment of the errant task.

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

8.8 HANDLING EXCEPTIONS IN-LINE

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

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

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

NOTE

If an invalid parameter causes an exceptional condition it should be handled by an exception handler. When using Nucleus system calls, the handler receives the parameter number of the first invalid parameter.

8.9 HANDLING EXCEPTIONS IN 80286 PROCESSOR SYSTEMS

The following sections are particularly important for users who are familiar with the iRMX 86 Operating System. The increased protection features of the 80286 microprocessor have resulted in a different way of handling exceptions and new exception codes.

The operating system software "catches" and returns most of the exceptional conditions. However, a few conditions occur because the microprocessor catches (or traps) an invalid condition. The trap causes control to be passed to special exception handling code which the iRMX II Operating System provides. This code examines the exception mode of the current task and acts in one of the following ways.

- It may call a system-supplied exception handler.
- It may call a user-supplied exception handler.
- It may return control directly to the faulting iRMX II task.

For 80286 processors, a CPU trap sets the instruction pointer (IP) register to point to the instruction that caused the CPU trap. This difference means that without an exception handler, an 80286-based application can never get past the instruction that caused the CPU trap. (Users familiar with 8086, 88, 186, or 188 processors will remember that with these processors a CPU trap sets the IP register to point to the instruction after the one that caused the CPU trap. This situation allows some applications to ignore errors and continue processing.)

Therefore, for 80286-based systems, you should always designate an exception handler to handle exception codes generated by CPU traps (programming errors), even if the handler does nothing more than increment the IP value that is pushed onto the task's stack when the trap occurs (that is, the return address). Without a handler of some kind, your application will get caught in an infinite loop in the event of a CPU trap.

If you use the exception handler supplied with the operating system to handle programmer errors, your application will run the same, regardless of the CPU. However, if you write your own exception handlers, you should include code to handle either of the situations mentioned in this section.

If you have user-written exception handlers that are being upgraded from the iRMX 86 Operating System to the iRMX II Operating System, be sure to change the code in exception handlers for the divide by zero trap. In the iRMX 86 Operating System after a divide by zero, the Nucleus returns to the next instruction whereas, in the iRMX II Operating System, the Nucleus returns to the same instruction. If you do not ensure that the return address is the next instruction, you could get caught up in an infinite loop. Intel recommends that you upgrade all user-written exception handlers to enable them to handle the new exception codes.

Table 8-2 lists the conditions which may cause a trap and the instruction to which it returns. The table also compares the handling of an iRMX II exception with an iRMX 86 exception (if you are operating the iRMX 86 Operating System on an 80286 processor). Some exceptions such as power failure do not return control to the faulting task. These exceptions have been marked N/A (not applicable).

Table 8-2. Return Address after an Exception

Interrupt Number	Description	Instruction Returned To iRMX II	iRMX 86 on 80286 Processor
0	Divide by zero	Same	Same
1	Single step	Next	Next
2	Power failure (non-maskable)	N/A	N/A
3	One byte interrupt instruction	Next	Next
4	Interrupt on overflow	Next	Next
5	Run time array bound error	Same	Same
6	Undefined opcode	Same	Same
7	NPX not present/NPX task switch	Same	Same
8	Double fault	N/A	N/A
9	NPX segment overrun	N/A	N/A
10	Invalid Task State Segment	N/A	N/A
11	Segment not present	Same	N/A
12	Stack exception	N/A	N/A
13	General Protection	Same	N/A
16	Processor Extension Error	N/A	Same

If your system supports an 80287 Numeric Processor Extension (NPX, some references to NDP in the error codes are for compatibility reasons) exceptions 7, 9 and 16 may be of special interest to you. Interrupt 7 may occur on systems that support an NPX and on those that don't. If your system does not have an NPX and you receive interrupt 7, treat it as you would any other program exception. However, if your system has an NPX and interrupt 7 occurs, do not try to service it as Interrupt 7 is reserved for the system. A context switch of the NPX environment takes place and the faulting task continues.

In the event of interrupts 9 or 16, the return address is that of the current instruction. However, the exception was caused by the previous NPX instruction. This is true because the 80287 NPX, unlike the 8087 NPX, does not cause an exception as soon as an error occurs. An exception occurs only when the next floating point instruction in the same task is executed.

8.10 SYSTEM CALLS FOR EXCEPTION HANDLERS

The following system calls manipulate exception handlers:

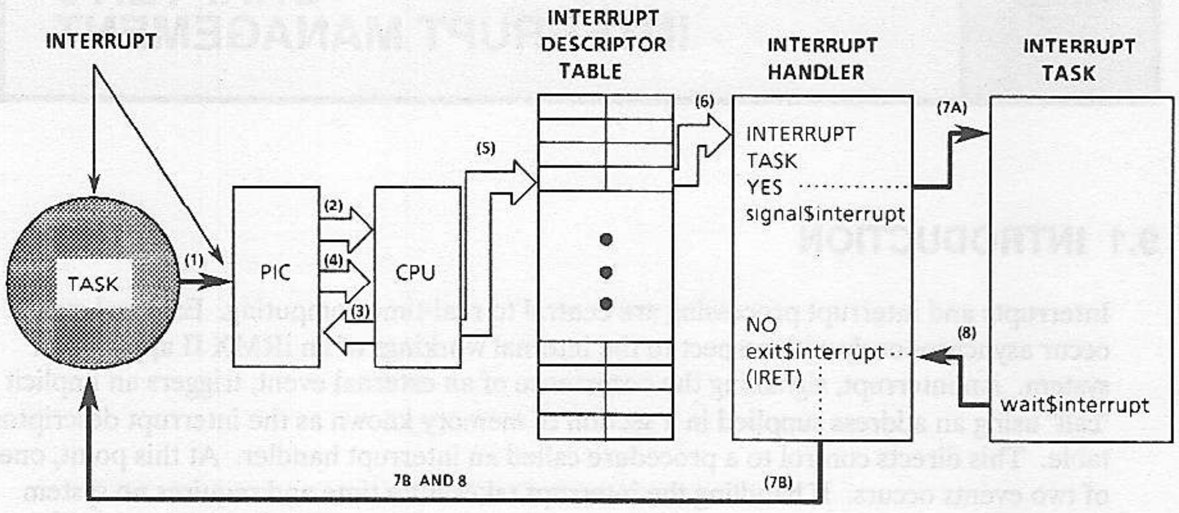
- SET$EXCEPTION$HANDLER--sets the exception handler and exception mode attributes of the calling task.
- GET$EXCEPTION$HANDLER--returns to the calling task the current values of its exception handler and exception mode attributes.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

CHAPTER 9 INTERRUPT MANAGEMENT

9.1 INTRODUCTION

Interrupts and interrupt processing are central to real-time computing. External events occur asynchronously with respect to the internal workings of an iRMX II application system. An interrupt, signalling the occurrence of an external event, triggers an implicit "call" using an address supplied in a section of memory known as the interrupt descriptor table. This directs control to a procedure called an interrupt handler. At this point, one of two events occurs. If handling the interrupt takes little time and requires no system calls other than certain interrupt-related system calls, the interrupt handler can process the interrupt itself. Otherwise, the interrupt handler can invoke an interrupt task, which deals with the interrupt. After the interrupt has been serviced by either the interrupt handler or the interrupt task, control returns to the ready application task with highest priority. See Figure 9-1 for a graphic representation of this interrupt process.



1. Programmable Interrupt Controller (PIC) receives an interrupt.
2. PIC signals the CPU.
3. CPU acknowledges the interrupt.
4. PIC sends the interrupt number to the CPU.
5. CPU obtains the interrupt handler from the interrupt descriptor table (IDT).
6. Control sent to the interrupt handler.

7a. Activate the interrupt task.

or

7b. Return to the interrupted task.

8. Return to the interrupted task.

Note: the solid arrows (➡) indicate software vectors;
the hollow arrows (⇒) indicate hardware vectors.

Figure 9-1. Interrupt Processing Model

9.2 INTERRUPT MECHANISMS

This section discusses the major concepts of interrupt processing: interrupt controllers and lines, interrupt levels, and the interrupt descriptor table. It also discusses assigning interrupt levels to external sources and disabling interrupts.

9.2.1 Interrupt Controllers and Interrupt Lines

External interrupts are passed through programmable interrupt controllers (PICs) such as the 8259A PIC. The iRMX II Operating System supports the configuration described here. Refer to the Extended *iRMX II Interactive Configuration Utility Reference Manual* for information on configuring the operating system to support the hardware configuration.

Under the iRMX II Operating System, interrupts must be funneled through 8259A PICs. In this environment, an individual master PIC can manage interrupts from as many as eight external sources. However, the iRMX II Operating System also supports an expanded (or cascaded) environment in which up to seven input lines of one master PIC are connected to slave PICs (one input line from the master PIC must be connected directly to the system clock). In a cascaded environment, an input line of a master PIC can connect either to an external interrupt or to a slave PIC, but not to both.

Because each of the slave PICs can manage eight interrupts, a cascaded environment allows the operating system to manage interrupts from as many as 56 external sources plus the system clock.

If your 80286-based system includes an 80287 NPX, you cannot connect the NPX to a PIC. Instead of using the PIC, the NPX uses CPU interrupt traps 7 and 16 to communicate directly with the 80286 component. Figure 9-2 illustrates this situation.

9.2.2 Interrupt Levels

The interrupt lines of the master and slave PICs are associated with numbers called interrupt levels, as shown in Figure 9-2. An interrupt level names an interrupt line and indicates the priority of the line (in general, the lower the number, the higher the priority). The interrupt lines on the master PIC are numbered M0 through M7. The interrupt lines on the slave PICs are numbered x0 through x7 (where x ranges from 0 to 7).

Lower-numbered interrupt lines like M0 or M1 (or lines from slave PICs connected to them) have higher priority than higher-level lines like M5 or M6 (or lines from slave PICs connected to them). Therefore, if two interrupts occur simultaneously, the PIC informs the CPU of the higher-priority interrupt first.

The Nucleus often disables low-priority interrupts to allow tasks to service high-priority interrupts. Refer to the "Disabling Interrupts" section of this chapter for more information.

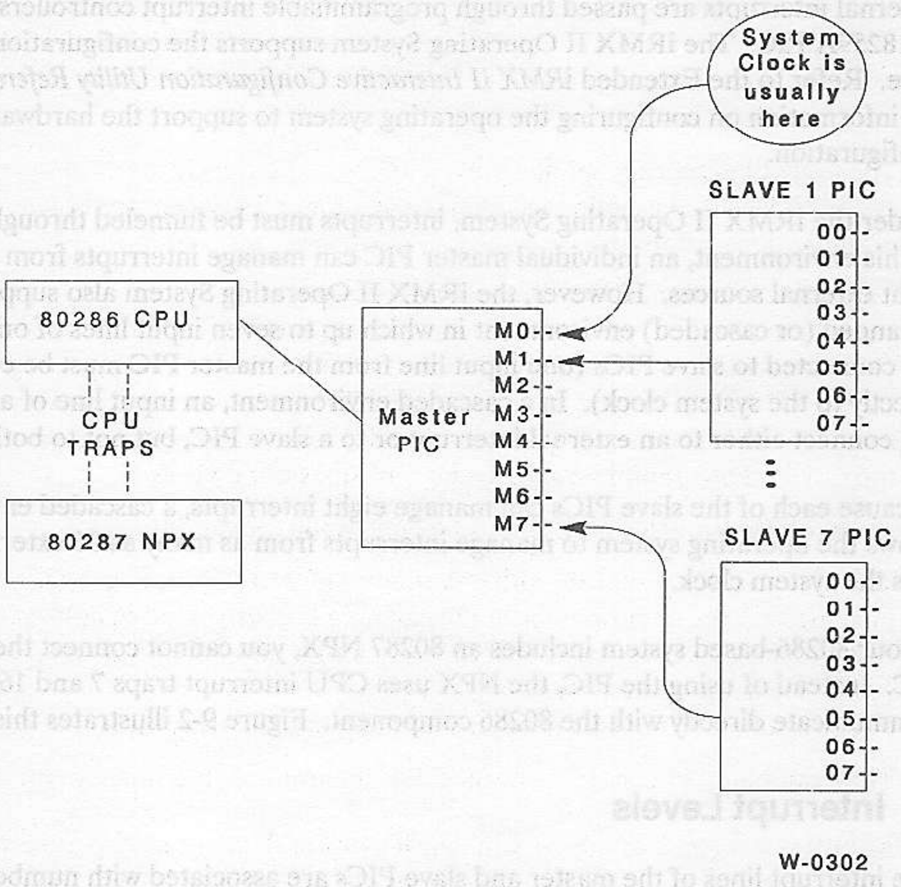


Figure 9-2. 80286 Interrupt Lines

9.2.3 Interrupt Descriptor Table

When an interrupt occurs, it triggers the processor to invoke a procedure whose address is listed in a section of memory called the interrupt descriptor table (IDT). You enter interrupt addresses into the IDT when configuring the system or dynamically, using SET$INTERRUPT. When an interrupt occurs, the processor uses the entry in the IDT as a pointer to the interrupt handling code to be executed for the specific interrupt. Each entry in the IDT is a descriptor that contains the physical address of the interrupt procedure that should be processed when the specified interrupt occurs. The IDT is similar to the GDT and LDT, except that it is referenced only as a result of an interrupt or a trap. The IDT may be located anywhere in the memory of the iRMX II Operating System. For more details about the IDT, see the *iAPX 286 Programmer's Reference Manual*.

Many different events may cause an interrupt. To allow the cause of the interrupt to be identified, the hardware assigns each interrupt cause a number and gives it an entry in the IDT. The IDT is composed of up to 256 entries, numbered 0-255. You specify the number of entries your application needs when you configure the system. Most users will not need more than 128 entries. If, for example, your system has only the 8259A PIC master with no 8259A PIC slaves, and does not use software interrupts, the first 64 entries are enough. The iRMX II Nucleus does not use entries 128-255. These entries are available for users. The entries are allocated as shown in Table 9-1.

Table 9-1. Allocation of Interrupt Entries

Entry Number	Description
0	divide by zero
1	single step (used by the iSDM monitor)
2	power failure (non-maskable interrupt,used by the iSDM monitor)
3	one-byte interrupt instruction (used by the iSDM monitor)
4	interrupt on overflow
5	run-time array bounds error
6	undefined opcode
7	NPX not present/NPX task switch
8	double fault
9	NPX segment overrun
10	invalid TSS
11	segment not present
12	stack exception
13	general protection
14-15	reserved
16	NPX error
17-55	reserved
56-63	8259A PIC master (external interrupts)
64-127	8259A PIC slaves (external interrupts)
128-255	unused

When an interrupt occurs on any master or slave level, the processor looks at the corresponding entry in the interrupt descriptor table to determine the address of the procedure to execute. The procedure that executes in response to an interrupt is called an interrupt handler.

For example, if a level M2 interrupt occurs, the processor examines interrupt descriptor 58 for the location of the interrupt handler for that level. Then it transfers control to the interrupt handler.

The Nucleus provides two system calls for setting up the interrupt descriptor table: SET$INTERRUPT and RESET$INTERRUPT. SET$INTERRUPT assigns an interrupt handler to an interrupt level by placing a pointer to the first instruction of the handler in the appropriate descriptor. RESET$INTERRUPT cancels the assignment of an interrupt handler by clearing out the appropriate entry in the interrupt descriptor table. With these two system calls, you can set up the descriptor table to meet your needs.

9.2.4 Assigning Interrupt Levels to External Sources

You must obey the following restrictions when assigning interrupt levels to external sources:

- You must assign the system clock to a master interrupt level. The level number is a configuration option and is described in the *Extended iRMX II Interactive Configuration Utility Reference Manual.*
- When you attach an interrupting device to a level on the master PIC, you cannot also attach a slave PIC to the same level. For example, suppose that you physically attach the device to level M3. This means that entry 59 (decimal) of the IDT must contain the address of the interrupt handler for the device. It also means that entries 88 through 95 (decimal) of the IDT (the slave level entries that correspond to master level M3) will not be used.

9.2.5 Disabling Interrupts

Occasionally you may want to prevent interrupt signals from causing an immediate interrupt. For example, you don't want low-priority interrupts to interfere with the servicing of a high-priority interrupt. In the iRMX II Operating System, each interrupt level can be <u>disabled</u>. In some circumstances (described later), the Nucleus disables levels. Tasks can also disable and enable levels by means of the DISABLE and ENABLE system calls. However, the master level reserved for the system clock should not be disabled or enabled.

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

An interrupt level can be disabled in four ways:

- A task can explicitly disable a specific interrupt level by invoking the DISABLE system call. Later, a task can re-enable the level by invoking the ENABLE system call.
- A task can invoke the SET$INTERRUPT system call to designate itself as the interrupt task for a particular interrupt level. When it makes this designation, the task

can specify a limit to the number of interrupts that it will queue. If enough interrupts occur on the task's interrupt level, the queue can become full. Whenever this happens, the operating system automatically disables the interrupt level until the queue ceases to be full.

- Whenever a task invokes the RESET$INTERRUPT system call to cancel the assignment of a particular interrupt handler to a particular interrupt level, the operating system automatically disables that interrupt level.
- To provide pre-emptive priority-based scheduling, the operating system can automatically disable or re-enable some interrupt levels whenever a task begins running, depending on the priority of the new running task and the priority of the previous running task. This allows high-priority tasks to run faster, without interrupts from lower-priority external devices. Table 9-2 shows the correlation between the levels disabled and the priority of the running task.

NOTE

A task that makes system calls when interrupts are disabled should never use the PL/M-286 DISABLE statement or the ASM286 CLI (clear interrupt-enable flag) instruction to disable operating system interrupts. Nucleus system calls may cause interrupts to be enabled.

Table 9-2. Interrupt Levels Disabled for Running Task

Task Priority	Disabled Levels	
	Slave Levels	Master Levels
0-2	00 - 77	M0 - M7
3-4	01 - 77	M1 - M7
5-6	02 - 77	M1 - M7
7-8	03 - 77	M1 - M7
9-10	04 - 77	M1 - M7
11-12	05 - 77	M1 - M7
13-14	06 - 77	M1 - M7
15-16	07 - 77	M1 - M7
17-18	10 - 77	M1 - M7
19-20	11 - 77	M2 - M7
21-22	12 - 77	M2 - M7
23-24	13 - 77	M2 - M7
25-26	14 - 77	M2 - M7
27-28	15 - 77	M2 - M7
29-30	16 - 77	M2 - M7
31-32	17 - 77	M2 - M7
33-34	20 - 77	M2 - M7
35-36	21 - 77	M3 - M7
37-38	22 - 77	M3 - M7
39-40	23 - 77	M3 - M7
41-42	24 - 77	M3 - M7
43-44	25 - 77	M3 - M7
45-46	26 - 77	M3 - M7
47-48	27 - 77	M3 - M7
49-50	30 - 77	M3 - M7
51-52	31 - 77	M4 - M7
53-54	32 - 77	M4 - M7
55-56	33 - 77	M4 - M7
57-58	34 - 77	M4 - M7
59-60	35 - 77	M4 - M7
61-62	36 - 77	M4 - M7
63-64	37 - 77	M4 - M7
65-66	40 - 77	M4 - M7
67-68	41 - 77	M5 - M7
69-70	42 - 77	M5 - M7
71-72	43 - 77	M5 - M7
73-74	44 - 77	M5 - M7

continued

Table 9-2. Interrupt Levels Disabled for Running Task (continued)

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

9.3 INTERRUPT HANDLERS AND INTERRUPT TASKS

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

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

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

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

9.3.1 Setting Up an Interrupt Handler

Interrupt handlers are generally written as PL/M-286 interrupt procedures, but they can be written in assembly language. If you use assembly language, you must save and restore all register values, as noted later.

Before an interrupt handler can service an interrupt level, a task must invoke the SET$INTERRUPT system call to bind the handler and an interrupt task to an interrupt level. SET$INTERRUPT operates as follows:

- One of the SET$INTERRUPT parameters, the interrupt$handler parameter, specifies the starting address of the interrupt handler. SET$INTERRUPT binds the handler to a level by placing this starting address into the IDT at the entry that corresponds to the level. When an interrupt of that level occurs, control automatically transfers through the IDT to the handler.
- Another parameter in SET$INTERRUPT, the interrupt$task$flag parameter, determines whether an interrupt task is associated with the level. If the interrupt$task$flag parameter contains a zero, there is no interrupt task for the specified level. Otherwise, the calling task becomes the interrupt task for the level.

If you want your interrupt handler to use another data segment, you can specify the selector of the interrupt handler's data segment in the interrupt$handler$ds parameter of SET$INTERRUPT. The interrupt handler can later load this value into the DS register by calling ENTER$INTERRUPT. Interrupt handlers written in PL/M-286 (including COMPACT model) have their DS registers loaded automatically on invocation. In most cases, an interrupt handler and an interrupt task are compiled together and share the same data areas.

When an iRMX II application system starts running, all interrupt levels are disabled. Before the operating system enables an interrupt level, a task must invoke SET$INTERRUPT. When SET$INTERRUPT binds an interrupt handler but not an interrupt task to a level, the operating system enables the level immediately. However, if SET$INTERRUPT binds the handler and an interrupt task to the level, the operating system does not enable the level until that task invokes the WAIT$INTERRUPT or RQE$TIMED$INTERRUPT system call (described later). An interrupt task should not enable its own level before making its first call to WAIT$INTERRUPT or RQE$TIMED$INTERRUPT.

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

9.3.2 Using an Interrupt Handler

If an interrupt handler services interrupts for a given level without invoking an interrupt task, the handler must assume one of two forms, depending on whether it requires the Nucleus to set up the selector of its data segment.

If the interrupt handler does not need to access the data segment, or if it can load the DS register with the data segment selector, then it should perform the following steps:

1. If in assembly language, save all register contents (PL/M does it for you when the procedure is given the INTERRUPT attribute).
2. Service the interrupt.
3. Call EXIT$INTERRUPT. (This sends an end-of-interrupt signal to the hardware.)
4. If in assembly language, restore all register contents.
5. Return.

In the rare case where you may want to use a special data segment, call ENTER$INTERRUPT immediately after step 1. An example of how to use EXIT$INTERRUPT is given in the *Extended iRMX II Nucleus System Calls Reference Manual.*

9.3.3 Using an Interrupt Task

If both an interrupt handler and an interrupt task are associated with a level, the interrupt handler invokes the interrupt task by making a SIGNAL$INTERRUPT system call. If a level has only an interrupt handler, however, the handler cannot call SIGNAL$INTERRUPT without causing an E$CONTEXT error.

9.3.3.1 Duties of the Interrupt Handler

If an interrupt handler invokes an interrupt task, the handler must perform the following steps:

1. If in assembly language, save the register contents.
2. Optionally, do some servicing.
3. Optionally, call ENTER$INTERRUPT.
4. Optionally, begin servicing the interrupt without system calls.
5. Call SIGNAL$INTERRUPT, which starts the interrupt task and enables higher (and possibly equal) priority interrupts.
6. Optionally, do some servicing.
7. If in assembly language, restore the register contents.
8. Return.

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

9.3.3.2 Duties of the Interrupt Task

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

```
Call SET$INTERRUPT;
     Do initialization;
     Do forever;
          Call WAIT$INTERRUPT (or RQE$TIMED$INTERRUPT);
          Service the interrupt (system calls allowed);
     End;
```

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

9.3.3.3 Interrupt Task Priorities

When a task becomes an interrupt task by calling SET$INTERRUPT, the Nucleus assigns a priority to it according to the interrupt level to be serviced. Table 9-3 shows the relationship between interrupt levels and the priorities of tasks that service those levels.

Table 9-3 lists several other values for each interrupt level. It lists the encoding for the interrupt level (the value used for the level parameter of SET$INTERRUPT), and the number of the corresponding IDT entry.

NOTE

If an interrupt task's priority exceeds the maximum priority attribute of its job, the Nucleus returns an exceptional condition code. Prevent this by giving the job a higher maximum priority.

Table 9-3. Interrupt Level and Task Priority Information

Encoding	Interrupt Level		IDT Slots	Interrupt Task Priority
	Master	Slave		
00H		00	64	4
01H		01	65	6
02H		02	66	8
03H		03	67	10
04H		04	68	12
05H		05	69	14
06H		06	70	16
07H		07	71	18
08H	M0		56	18
10H		10	72	20
11H		11	73	22
12H		12	74	24
13H		13	75	26
14H		14	76	28
15H		15	77	30
16H		16	78	32
17H		17	79	34
18H	M1		57	34
20H		20	80	36
21H		21	81	38
22H		22	82	40
23H		23	83	42
24H		24	84	44
25H		25	85	46
26H		26	86	48
27H		27	87	50
28H	M2		58	50
30H		30	88	52
31H		31	89	54
32H		32	90	56
33H		33	91	58
34H		34	92	60
35H		35	93	62

(continued)

Table 9-3. Interrupt Level and Task Priority Information (continued)

Encoding	Interrupt Level		IDT Slots	Interrupt Task Priority
	Master	Slave		
36H		36	94	64
37H		37	95	66
38H	M3	59		66
40H		40	96	68
41H		41	97	70
42H		42	98	72
43H		43	99	74
44H		44	100	76
45H		45	101	78
46H		46	102	80
47H		47	103	82
48H	M4		60	82
50H		50	104	84
51H		51	105	86
52H		52	106	88
53H		53	107	90
54H		54	108	92
55H		55	109	94
56H		56	110	96
57H		57	111	98
58H	M5		61	98
60H		60	112	100
61H		61	113	102
62H		62	114	104
63H		63	115	106
64H		64	116	108
65H		65	117	110
66H		66	118	112
67H		67	119	114
68H	M6		62	114
70H		70	120	116
71H		71	121	118
72H		72	122	120
73H		73	123	122
74H		74	124	124
75H		75	125	126
76H		76	126	128
77H		77	127	130
78H	M7		63	130

9.3.4 Interrupt Servicing Patterns

Figure 9-3 illustrates the relationships between the servicing patterns of interrupt handlers and interrupt tasks.

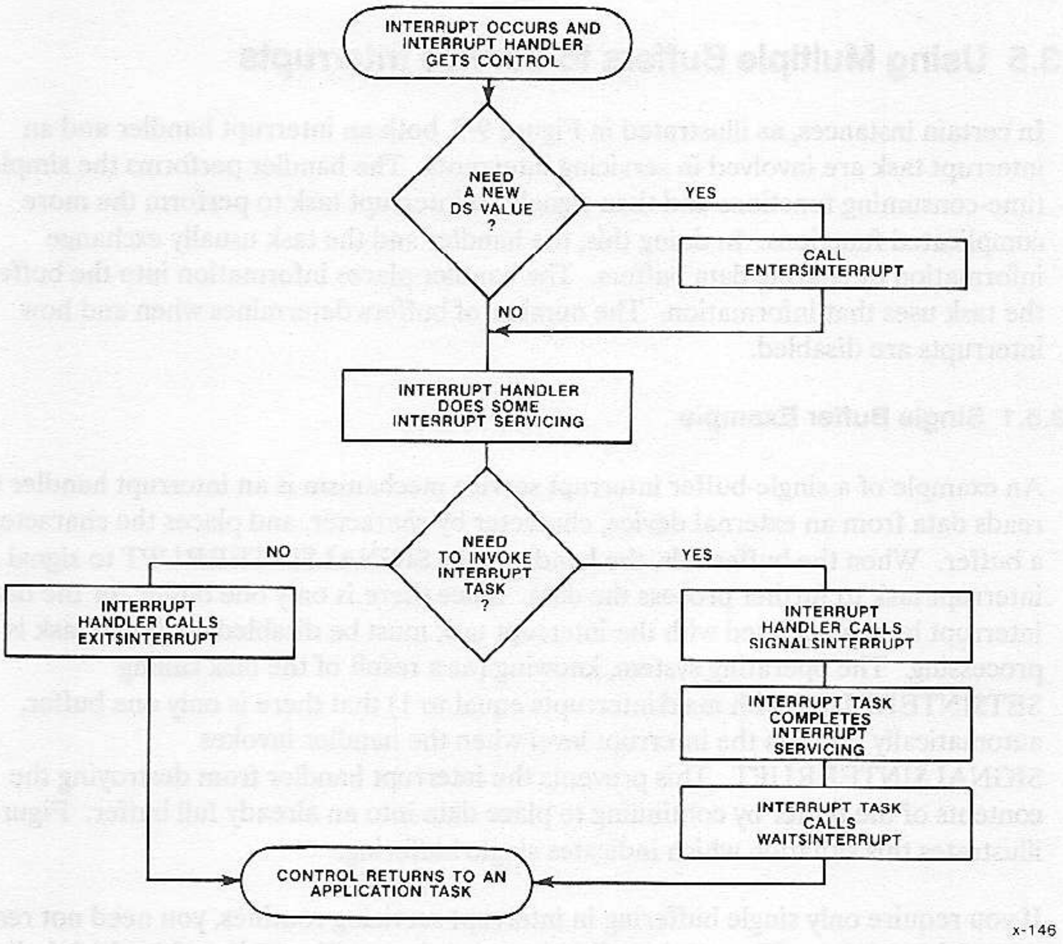

INTERRUPT OCCURS AND INTERRUPT HANDLER GETS CONTROL
NEED A NEW DS VALUE ?
YES
CALL ENTER$INTERRUPT
NO
INTERRUPT HANDLER DOES SOME INTERRUPT SERVICING
NEED TO INVOKE INTERRUPT TASK ?
NO
YES
INTERRUPT HANDLER CALLS EXIT$INTERRUPT
INTERRUPT HANDLER CALLS SIGNAL$INTERRUPT
INTERRUPT TASK COMPLETES INTERRUPT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

Figure 9-3. Flow Chart of Interrupt Handling

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

9.3.5 Using Multiple Buffers to Service Interrupts

In certain instances, as illustrated in Figure 9-3, both an interrupt handler and an interrupt task are involved in servicing interrupts. The handler performs the simple, less time-consuming functions and then signals an interrupt task to perform the more complicated functions. In doing this, the handler and the task usually exchange information by sharing data buffers. The handler places information into the buffers and the task uses that information. The number of buffers determines when and how interrupts are disabled.

9.3.5.1 Single Buffer Example

An example of a single-buffer interrupt service mechanism is an interrupt handler that reads data from an external device, character by character, and places the characters into a buffer. When the buffer fills, the handler calls SIGNAL$INTERRUPT to signal an interrupt task to further process the data. Since there is only one buffer for the data, the interrupt level associated with the interrupt task must be disabled while the task is processing. The operating system, knowing (as a result of the task calling SET$INTERRUPT with max$interrupts equal to 1) that there is only one buffer, automatically disables the interrupt level when the handler invokes SIGNAL$INTERRUPT. This prevents the interrupt handler from destroying the contents of the buffer by continuing to place data into an already full buffer. Figure 9-4 illustrates this situation which indicates single buffering.

If you require only single buffering in interrupt servicing routines, you need not read the rest of this section. (Ensure that all interrupt tasks specify a value of 1, which indicates single buffering, for the interrupt$task$flag parameter in the call to SET$INTERRUPT.)

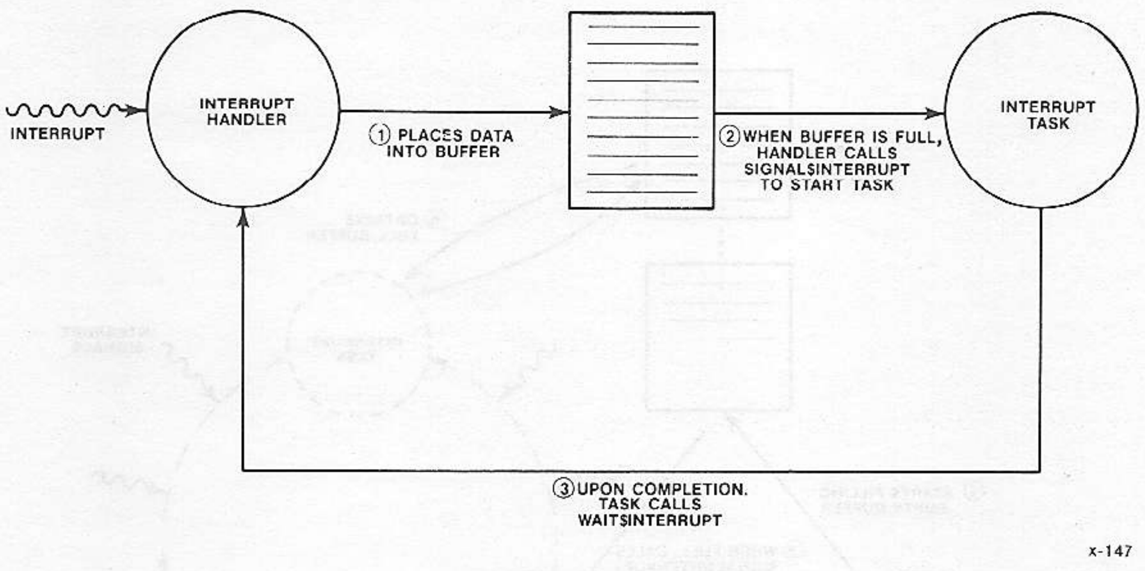

INTERRUPT
INTERRUPT HANDLER
① PLACES DATA INTO BUFFER
② WHEN BUFFER IS FULL, HANDLER CALLS SIGNAL$INTERRUPT TO START TASK
INTERRUPT TASK
③ UPON COMPLETION. TASK CALLS WAIT$INTERRUPT
x-147

Figure 9-4. Single-Buffer Interrupt Servicing

9.3.5.2 Multiple Buffer Example

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

While this occurs, the interrupt task processes the full buffer. When the task completes the processing, it calls WAIT$INTERRUPT or RQE$TIMED$INTERRUPT to indicate that it is ready to accept another SIGNAL$INTERRUPT request (another full buffer) and to indicate that the buffer it just finished processing is available for reuse by the handler. Figure 9-5 illustrates this multiple buffer situation.

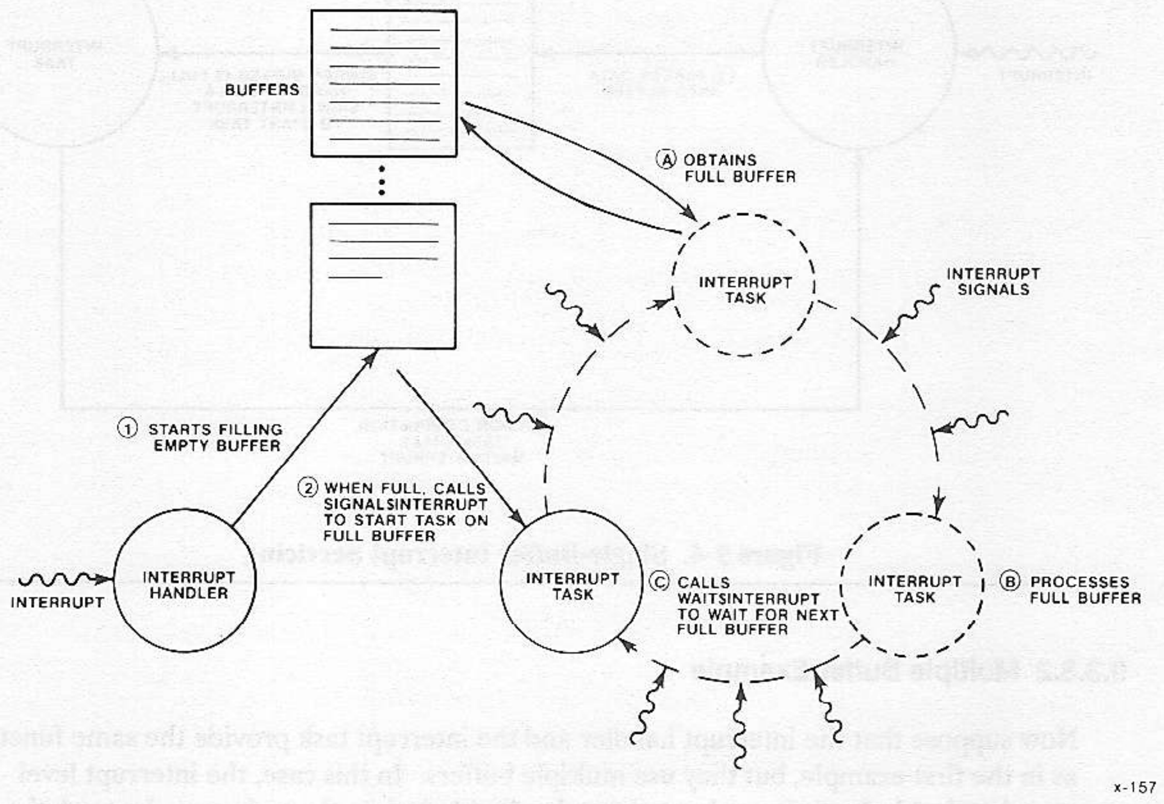

BUFFERS
Ⓐ OBTAINS FULL BUFFER
INTERRUPT TASK
INTERRUPT SIGNALS
① STARTS FILLING EMPTY BUFFER
② WHEN FULL, CALLS SIGNAL$INTERRUPT TO START TASK ON FULL BUFFER
INTERRUPT
INTERRUPT HANDLER
INTERRUPT TASK
Ⓒ CALLS WAIT$INTERRUPT TO WAIT FOR NEXT FULL BUFFER
INTERRUPT TASK
Ⓑ PROCESSES FULL BUFFER
x-157

Figure 9-5. Multiple-Buffer Interrupt Servicing

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

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

9.3.5.3 Specifying The Count Limit

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

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

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

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

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

NOTE

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

Table 9-4 illustrates the situation described in this section. It shows the actions of the handler and the task illustrated in Figure 9-4. The table is broken up into three parts: actions of the interrupt handler, actions of the interrupt task, and SIGNAL$INTERRUPT count. The count limit is set to two. The table shows the actions of both the handler and the task through time, and the change in value of the count.

Table 9-4 documents two extreme conditions, labeled A and B. At position A, the interrupt handler fills its last available buffer and calls SIGNAL$INTERRUPT to notify the task. However, the task has not finished processing the first buffer, so the count is incremented to the limit and interrupts are disabled until the task finishes with the first buffer and calls WAIT$INTERRUPT.

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

Table 9-4. Handler and Task Interaction through Time

Time	Interrupt Handler	Interrupt Task	SIGNAL$ INTERRUPT Count
		Call SET$INTERRUPT to establish handler and task for level, setting count limit to 2.	0
		Call WAIT$INTERRUPT to wait for first request from handler.	0
Intrpt	Process interrupt, start filling first buffer.		
Intrpt	Process interrupt, continue filling first buffer.		
Intrpt	Process interrupt. Buffer is full. Call SIGNAL$INTERRUPT.		
		Start processing first full buffer.	1
Intrpt	Process interrupt. Start filling next buffer.		
Intrpt	Process interrupt. Buffer is full. Call SIGNAL$INTERRUPT. Count is at limit. Interrupt level is disabled.		2

(continued)

Table 9-4. Handler and Task Interaction through Time (continued)

		Call WAIT$INTERRUPT. Task starts processing next full buffer immediately and returns empty buffer. Interrupt level is enabled.	1
Intrpt	Process interrupt. Start filling next buffer.		
		Call WAIT$INTERRUPT. No full buffers are available. Task waits for next request.	0
Intrpt	Process interrupt. Buffer is full. Call SIGNAL$INTERRUPT.		
		Start processing next full buffer.	1

9.3.5.4 Enabling Interrupt Levels From Within a Task

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

To prevent this situation, the interrupt task can invoke the ENABLE system call immediately after it processes the buffer, enabling its associated interrupt level. This means that while the task engages in its time-consuming activities, the interrupt handler can accept further interrupts and place the data into the buffer just released by the task. (You can also use this technique whenever the count limit is one, whether or not you use a buffer.)

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

- The count of outstanding SIGNAL$INTERRUPT requests is incremented, causing it to exceed the user-specified limit.
- An exception code, E$INTERRUPT$OVERFLOW, is returned to the interrupt handler to indicate this overflow.

- The interrupt level is again disabled. The interrupt task cannot explicitly enable the level again until the count falls to or below the limit.

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

If a task other than an interrupt task tries to enable the level, one of three events may occur:

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

9.4 HANDLING SPURIOUS INTERRUPTS

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

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

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

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

The following sections describe several options for isolating spurious interrupts.

9.4.1 Calling GET$LEVEL

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

This method is simple to implement, but it does take more (handler) time to execute GET$LEVEL. Some handlers may have speed requirements that prohibit the use of GET$LEVEL.

9.4.2 Judicious Selection of Interrupt Levels

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

9.4.3 Examining the In-Service Register

Another way that a level 7 interrupt handler can check for spurious interrupts is by immediately reading the ISR (In-Service Register) of the corresponding PIC. If the BYTE value obtained from that register does not have a 1 in the high-order bit, the interrupt is spurious. To read the value, the handler must know the port address of the ISR. In PL/M-286, the following lines perform this check when placed at the beginning of the interrupt handler:

```
IF ((INPUT (port address of ISR)) AND 80H) = 0
    THEN interrupt is spurious
```

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

9.5 EXAMPLES OF INTERRUPT SERVICING

Tables 9-5, 9-6, and 9-7 should help you understand the major points already described. Each table outlines the turning points in a scenario where an interrupt handler is assigned to a level, an interrupt arrives at that level and is serviced, and the assignment of an interrupt handler is cancelled. The tables show the following cases:

- Table 9-5--the interrupt handler deals with the interrupt (handler is assigned to master level 4).
- Table 9-6--the interrupt handler invokes an interrupt task, either immediately or after filling a single buffer of data (handler is assigned to master level 4).
- Table 9-7--an interrupt handler and an interrupt task use multiple buffers to service interrupts (handler is assigned to slave level 35).

In the right-hand column of each table, the phrase "interrupt levels necessarily disabled" indicates that the events of the example cause certain levels to be enabled or disabled. Other events, outside the scope of the example, might cause other levels to be disabled as well.

Table 9-5. Servicing Interrupts with an Interrupt Handler

Step	Events	Explanation	Interrupt Levels Necessarily Disabled
1	-	No interrupt handler assigned to level M4.	M4
2	RQ$SET$INTERRUPT (LEVEL$4,0,...);	A task assigns an interrupt handler to level M4.	None
3	Level 4 device interrupts	An interrupt arrives at level M4.	All
4	. . .	The interrupt is serviced by the interrupt handler.	All
5	RQ$EXIT$INTERRUPT (LEVEL$4,...);	Interrupt hardware reset by the interrupt handler.	All
6	Interrupt handler returns.	Interrupts are re-enabled.	None
7	RQ$RESET$INTERRUPT (LEVEL$4,...);	A task cancels the assignment of an interrupt handler to level M4.	M4

Table 9-6. Servicing Interrupts with an Interrupt Task

Step	Events	Explanation	Interrupt Levels Necessarily Disabled
1	-	No interrupt handler assigned to level M4.	M4
2	RQ$SET$INTERRUPT (LEVEL$4, 1, ...);	A task assigns an interrupt handler to level M4 and assigns itself to be the interrupt task for that level. It specifies that one SIGNAL$INTERRUPT request can be outstanding.	M4-M7, 50-77
3	RQ$WAIT$INTERRUPT or RQE$TIMED$-INTERRUPT (LEVEL$4,...);	The interrupt task begins to wait for an interrupt.	None
4	Level 4 device interrupts	An interrupt arrives at level M4. The interrupt handler gains control and optionally, does some servicing. The handler may service several interrupts by performing steps 4 through 6 of Table 9-5.	All
5	RQ$SIGNAL$INTERRUPT (LEVEL$4,...);	The interrupt handler invokes the interrupt task.	M4-M7, 50-77
6	. . .	The interrupt is serviced by the interrupt task.	M4-M7, 50-57
7	RQ$WAIT$INTERRUPT or RQE$TIMED$-INTERRUPT (LEVEL$4,...);	The interrupt task finishes and begins to wait for another level M4 interrupt. Control passes back to the interrupt handler and then back to an application task.	None
8	RQ$RESET$INTERRUPT (LEVEL$4,...);	A task cancels the assignment of a handler to M4.	M4

Table 9-7. Servicing Interrupts with an Interrupt Handler, an Interrupt Task, and Multiple Buffering

Step	Events	Explanation	Interrupt Levels Necessarily Disabled
1	-	No interrupt handler assigned to level 35.	35
2	RQ$SET$INTERRUPT (LEVEL$35, 2, ...);	A task assigns an interrupt handler to level 35 and assigns itself to be the interrupt task for that level. It specifies two SIGNAL$INTERRUPT requests can be outstanding (double buffering).	M4-M7, 36-77
3	RQ$WAIT$INTERRUPT or RQE$TIMED$-INTERRUPT (LEVEL$35,...);	The interrupt task begins to wait for an interrupt.	None
4	Level 35 device interrupts	An interrupt arrives at level 35. The interrupt handler gains control and does some servicing.	All
5	. . .	The handler services all interrupts, as described in steps 4 through 6 of Table 9-5, until the first buffer is full.	All
6	RQ$SIGNAL$INTERRUPT (LEVEL$35,...);	The interrupt handler invokes the interrupt task.	M4-M7, 36-77

continued

Table 9-7. Servicing Interrupts with an Interrupt Handler, an Interrupt Task, and Multiple Buffering (continued)

Step	Events	Explanation	Interrupt Levels Necessarily Disabled
7	. . .	The interrupt task processes the full buffer. Meanwhile, the interrupt handler services interrupts, as described in steps 4 through 6 of Table 9-6, until the next buffer is full.	M4-M7, 36-77
8	RQ$WAIT$INTERRUPT or RQE$TIMED$-INTERRUPT (LEVEL$35,...);	The interrupt task finishes and waits for another signal from the interrupt handler. Control passes back to the interrupt handler and then back to an application task.	None
9	RQ$RESET$INTERRUPT (LEVEL$35,....);	A task cancels the assignment of an interrupt handler to level 35.	35

9.6 SYSTEM CALLS FOR INTERRUPTS

The following system calls manipulate interrupts:

- SET$INTERRUPT--assigns an interrupt handler and, if desired, an interrupt task to an interrupt level.
- RESET$INTERRUPT--cancels the assignment made to a level by SET$INTERRUPT and, if applicable, deletes the interrupt task for that level.
- EXIT$INTERRUPT--used by interrupt handlers to send an end-of-interrupt signal to hardware.
- SIGNAL$INTERRUPT--used by interrupt handlers to invoke interrupt tasks.
- RQE$TIMED$INTERRUPT--puts the calling interrupt task to sleep for a specified time. The task awakens either when the specified time elapses or a SIGNAL$INTERRUPT system call is issued.
- WAIT$INTERRUPT--suspends the calling interrupt task until it is called into service by an interrupt handler (via SIGNAL$INTERRUPT).

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

For a complete list of the iRMX II Nucleus system calls, see the *Extended iRMX II Nucleus System Calls Reference Manual.*

Intel®

CHAPTER 10 OPERATING SYSTEM EXTENSIONS

10.1 INTRODUCTION

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

Material presented in this chapter is intended for programmers who write system programs to extend the operating system. Users familiar with the iRMX I Operating System should read this chapter carefully as the method of implementing operating system extensions in the iRMX II Operating System is different.

10.2 THREE WAYS OF ADDING FUNCTIONALITY

If more than one job in your application system requires a function not supplied by the iRMX II 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 LIB286. After compiling each job that requires the function, use BND286 to bind 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 interface.

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

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

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

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

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

10.3 Creating an Operating System Extension

Creating an OS extension involves writing several procedures and establishing entry points or call-gates for them.

10.3.1 Procedures Used In Operating System Extensions

Every OS extension is composed of an interface and a function procedure. Figure 10-2 illustrates the simplest arrangement of these functions.

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.

A third kind of procedure may also be employed, however, it is optional.

Entry Procedure

The entry procedure serves as a multiplexor for OS extensions supporting more than one system call. Figure 10-1 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. Users familiar with the iRMX I Operating System should note that entry procedures are less important in the iRMX II Operating System because there are now 8K GDT slots in which to put the extensions rather than 32 software interrupts.

Figure 10-2 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 application 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 a call-gate.

Call-gates are used to enter the OS extensions. They redirect flow within a task from one code segment to another. Each system call uses a call-gate to transfer the program directly to the iRMX service routine requested. This makes it possible to go directly from the interface procedure to the function procedure. Call-gates are part of the descriptor tables and can be reserved when you configure the system. Since there are 8K slots in the GDT, you have a great deal of flexibility when creating operating system extensions.

Figure 10-3 contains, in algorithmic form, summaries of these descriptions. Also, Chapter 11 contains an example of an OS extension that manages a customized object type.

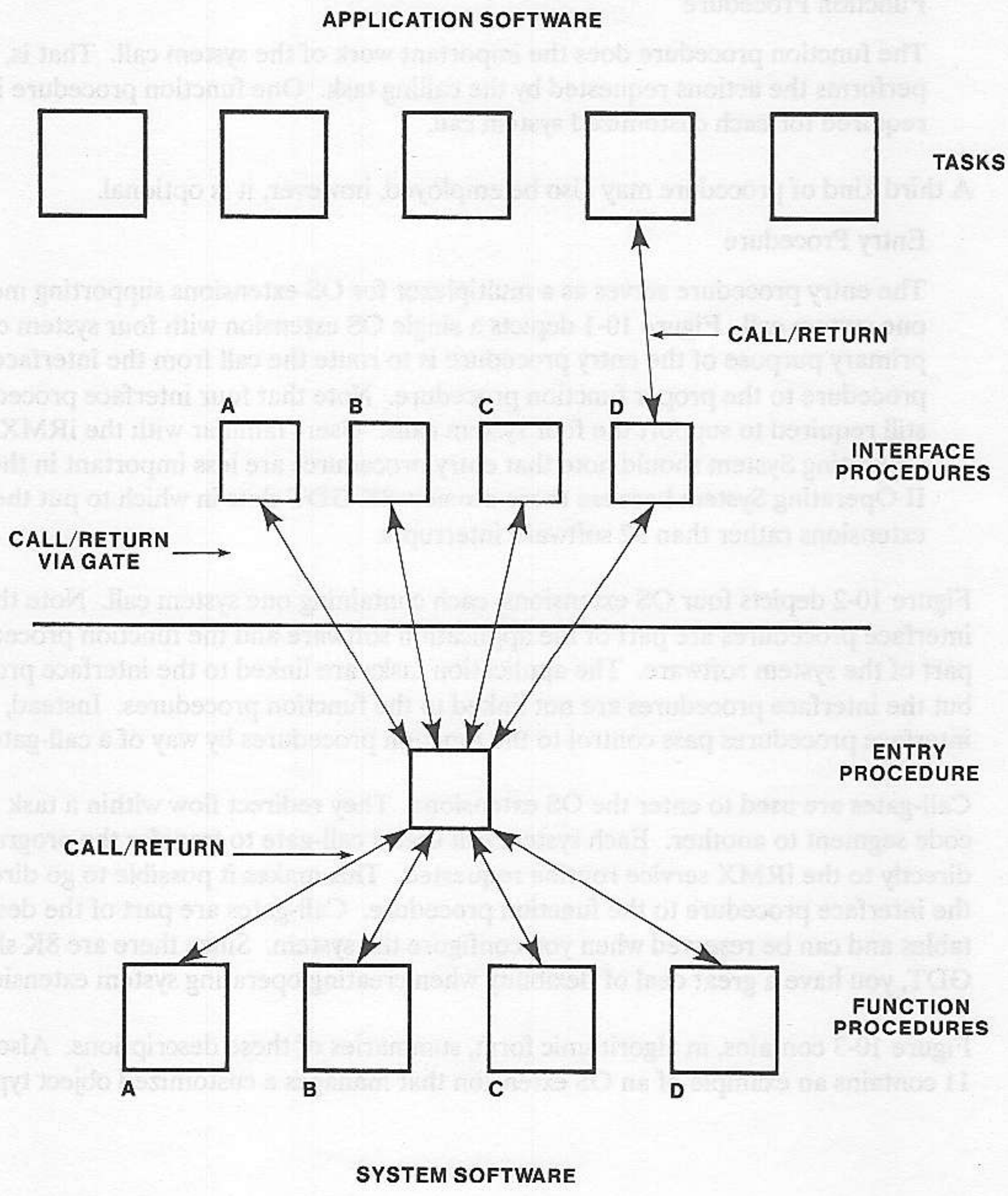


Figure 10-1. OS Extensions with Entry Procedure

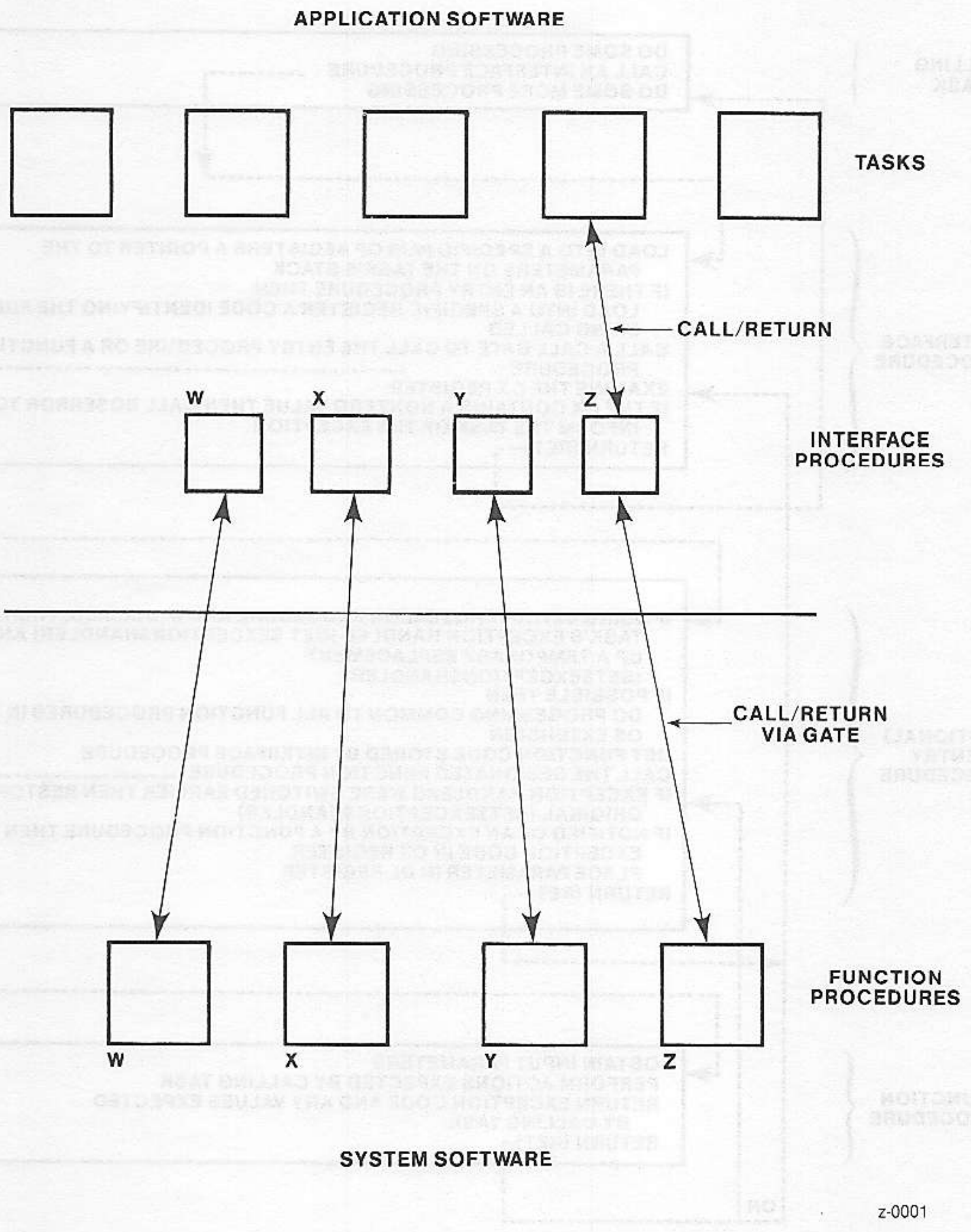
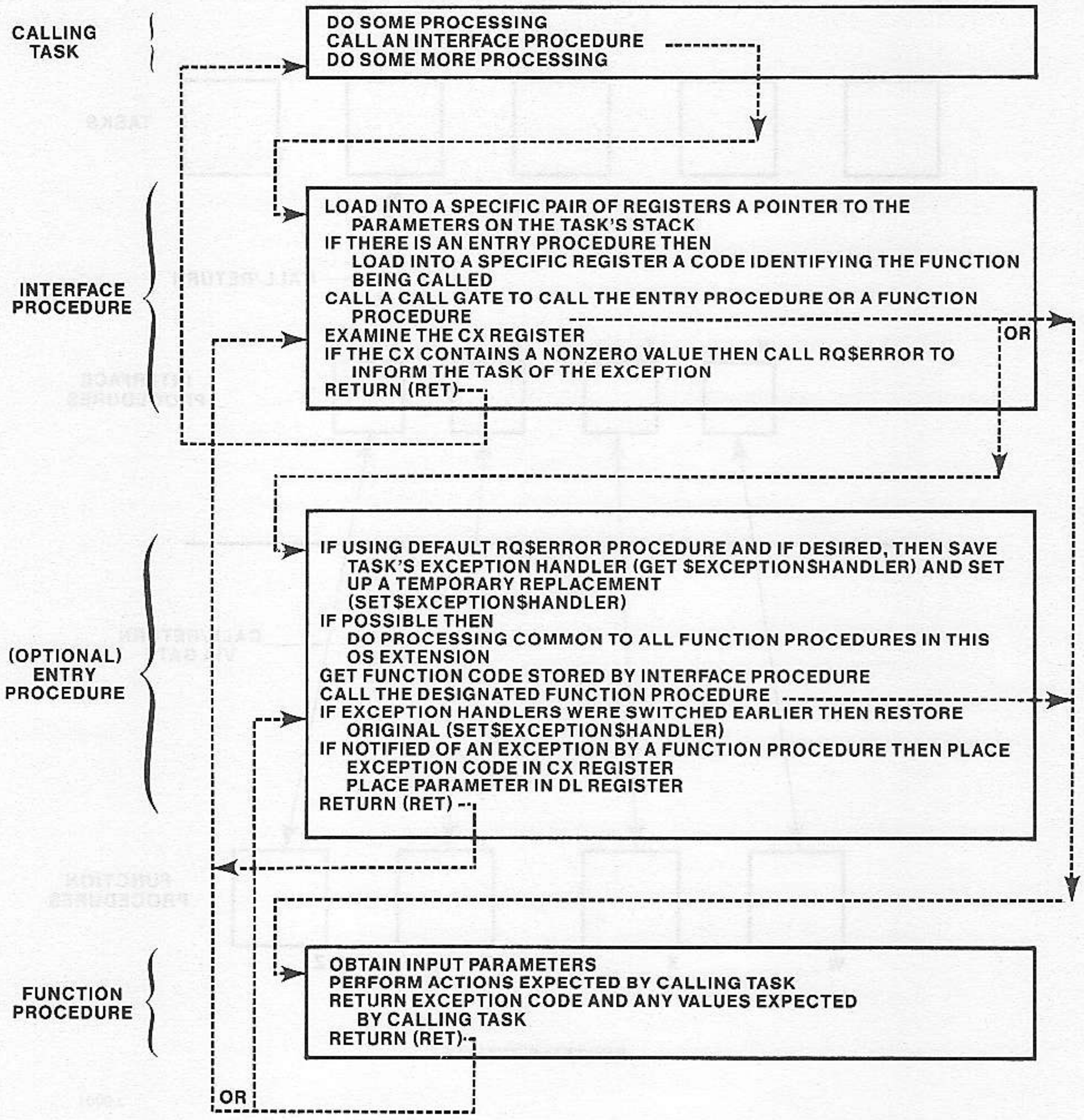


Figure 10-2. OS Extension without Entry Procedure



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

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

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

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

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

10.3.2 Interface Procedures

For each system call in your OS extension, you must write a re-entrant assembly language interface procedure. (For detailed information concerning the ASM286 Assembly Language, refer to the *ASM286 Assembly Language Reference Manual.*) This procedure uses a call-gate to transfer control from the task that invoked the system call to a function procedure. When transferring control to a function procedure whose call-gate number is 441H, for example, the interface procedure calls GATE_0441 which is the PUBLIC name for this gate. (You can find a gate's PUBLIC name in the MP2 file generated by BLD286.)

A second important function of the interface procedure is informing the calling task (or its exception handler) of any exceptional conditions that have occurred. The function procedure 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.

Calls a procedure named RQ$ERROR, if an exceptional condition exists.

The Nucleus interface library contains a default RQ$ERROR procedure or you may write your own RQ$ERROR procedure (further details are given in section, "RQ$ERROR Procedure").

Another important function of interface procedures is that they make function procedures independent of the PL/M-286 model being used to compile your application. This is done by providing a library of interface procedures for each PL/M-286 model. The benefit of this independence is that only one call-gate, and its related function procedure, is needed for each additional application function. The call-gate and its function procedure are then available for use by all PL/M-286 models.

10.3.3 Entry Procedures

Each OS extension comprising more than one system call may include a reentrant entry procedure, whose purpose is to route the call to the appropriate function procedure. As stated previously, this procedure is optional in the iRMX II Operating System because there are 8K GDT slots in which to put extensions.

Other possible functions of entry procedures are

To set up the exception handling mechanism for the OS extension, if this option is required (see below).

To perform a routine common to all system calls in this OS extension.

To transmit the exception incurred by the function procedure back to the interface routine--in CX and DL registers as explained above.

Write the entry procedure in assembly language so that you can directly access the stack and the registers. This gives you access to the input parameters passed by the calling task and the interface procedure. It also allows you to set the CX and DL registers in the event of an exceptional condition. To enable the entry procedure to route the call to the appropriate function procedure, the interface procedure must send a code identifying the function procedure called by the entry procedure. The interface procedure does this by loading the code into a previously designated register or onto the stack of the calling task.

10.3.4 Function Procedures

The duties of the function procedure are principally to perform the actions requested by the calling task. If there is no entry procedure, the function procedure should inform the interface procedure of the system call's exception status. It does this by setting CX and DL, as described in the description of entry procedures. Function procedures should be reentrant and can be written in PL/M-286 or assembly language.

10.3.5 RQ$ERROR And NUC$ERROR Procedures

The iRMX II Operating System has one interface library, RMXIFC (or RMXIFL depending on the segmentation model) which contains both the RQ$ERROR and NUC$ERROR procedures. These procedures invoke the task's exception handler. NUC$ERROR is a procedure called by the Nucleus interface procedures when an exceptional condition occurs, and RQ$ERROR is a procedure called by the interface procedures of all the other subsystems. For example, if your application task makes a SEND$MESSAGE system call to a non-existent mailbox, the Nucleus returns the error, in the CX and DL registers, to the Nucleus interface library linked to your application task. The procedure in the library then calls NUC$ERROR to process the error.

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

RQ$ERROR gets the exception code and parameter number from the CX and DL registers and then makes a SIGNAL$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$ERROR restores CX and DL with the exception code and parameter number and places a value of 0FFFFH in the AX register. This version of RQ$ERROR should be linked to application tasks to ensure that their exception handlers are called when exceptional conditions occur. Figure 10-4 illustrates the flow of control from an application task to an exception handler when the task incurs an exception.

NUC$ERROR performs the same functions as RQ$ERROR. However, it does not call SIGNAL$EXCEPTION. Instead, when a Nucleus system call returns with an exceptional condition, the stack contains three (3) extra words to process the exception. One word is the exception mode and the other two contain a pointer to the exception handler. If the exception mode indicates the need to call the exception handler, NUC$ERROR calls the exception handler directly.

If you do not want to use the default RQ$ERROR or NUC$ERROR procedures, you can write your own procedures. 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$ERROR procedure is one that simply places 0FFFFH in AX and then issues a RETURN, returning control directly to the application task to avoid the task's normal exception handler. If you write your own NUC$ERROR you must always pop three extra words from the stack.

To ensure that your own procedure is called instead of the default version, 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.

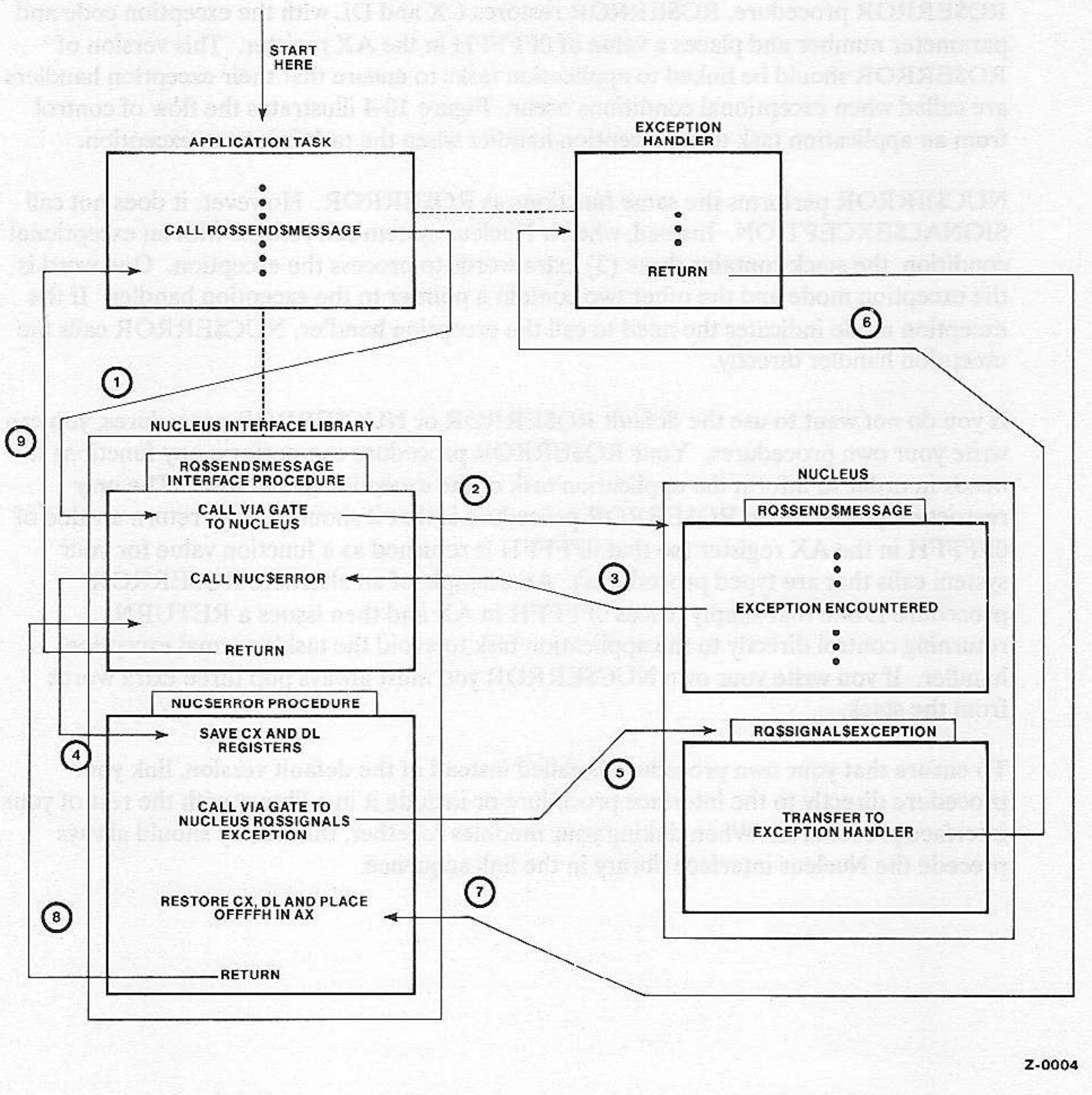


Figure 10-4. Handling Exceptions with an Exception Handler

10.4 ESTABLISHING EXCEPTION-HANDLING MECHANISMS

The following section refers to the OS extension code and the interface it uses to call existing operating system calls (Nucleus, BIOS, etc.). The interface discussed here is not the interface (described at the beginning of this chapter) used by the application task to call the OS extension functions.

Exception handling in an OS extension 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. Using any other method results in unpredictable execution of the OS extension code. If an exception occurs while the extension calls an iRMX II system call, the flow within the exception code is dependent on the exception handler and mode of the task invoking the extension.

If the OS extension has its own exception handler, the function procedure must change the exception handler from that of the calling task to an exception handler for the OS extension. To make this change, the function procedure should first call 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. Just before returning control to the interface, the function procedure again calls SET$EXCEPTION$HANDLER to restore the original values. In case of an entry procedure, the entry procedure saves and restores the exception handler and mode.

If you want the OS extension to handle exceptions in-line, you can follow the above strategy, calling SET$EXCEPTION$HANDLER with the EXCEPTION$MODE parameter set to NEVER. This is the simplest and most straightforward method. However, it costs three (3) Nucleus calls (to get, set, and restore) for every extension call. This is because it is done upon entry and exit from the function procedure.

Another way of handling exceptions in-line is to link your OS extension to your version of RQ$ERROR or NUC$ERROR. The RQ$ERROR procedure may simply 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.

If you want to override NUC$ERROR with your own procedure, you should return from your version of the NUC$ERROR procedure by using RET 6, to pop three (3) extra words from the stack. These words are used by the Nucleus to save the call to RQ$SIGNAL$EXCEPTION.

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 means that the function procedure of your OS extension places the condition code and parameter number in CX and DL, and returns to the interface linked to the application task. The interface procedure then calls RQ$ERROR in the event of an exceptional condition. The RQ$ERROR procedure that gets called is the one in the interface library linked to the calling task, not the one in the interface library linked to the OS extension.

Figure 10-5 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. When examining the diagram follow the numbered arrows. 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 to the OS extension. The RQ$ERROR procedure linked to the application task is the default procedure which calls SIGNAL$EXCEPTION.

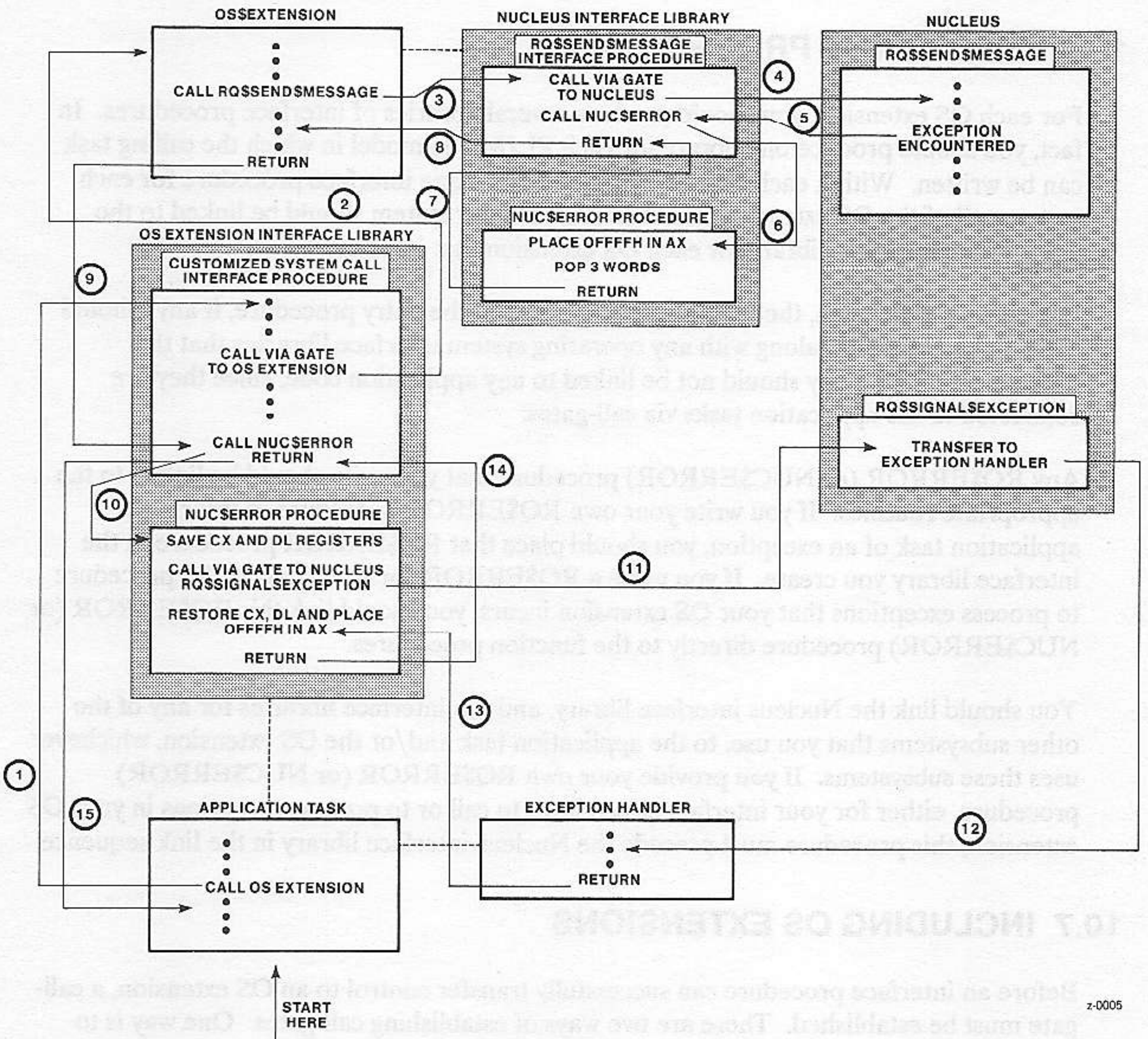


Figure 10-5. Control Flow for Handling Exceptions In-Line

10.5 CUSTOMIZED EXCEPTION CODES

When adding OS extensions, you may want to add your own 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).

10.6 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-286 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 module in your system should be linked to the appropriate interface library for each OS extension that is called.

For each OS extension, the function procedure (and the entry procedure, if any) 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 call-gates.

Any RQ$ERROR (or NUC$ERROR) procedure that you write should be linked to the appropriate routines. If you write your own RQ$ERROR procedure to inform the application task of an exception, you should place that RQ$ERROR procedure in the interface library you create. If you write a RQ$ERROR (or NUC$ERROR) procedure to process exceptions that your OS extension incurs, you should link this RQ$ERROR (or NUC$ERROR) procedure directly to the function procedures.

You should link the Nucleus interface library, and the interface libraries for any of the other subsystems that you use, to the application task and/or the OS extension, whichever uses these subsystems. If you provide your own RQ$ERROR (or NUC$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.

10.7 INCLUDING OS EXTENSIONS

Before an interface procedure can successfully transfer control to an OS extension, a call-gate must be established. There are two ways of establishing call-gates. One way is to include them dynamically using the system call RQE$SET$OS$EXTENSION. If you use RQE$SET$OS$EXTENSION to designate the call-gate through which your OS extension is to be entered, you must specify the gate number when you configure the system. Then when you invoke the system call, enter the gate number and the start address of the first instruction.

OS extensions can also be configured into your system by using the "OS Extension" Screen of the Interactive Configuration Utility (ICU). In this case, the Nucleus initializes your OS extensions. For more information see the *iRMX II Interactive Configuration Utility Reference Manual.*

10.8 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 only 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.

None of these system calls--DISABLE$DELETION, ENABLE$DELETION, and FORCE$DELETE--should be used in jobs that an operator invokes via a Human Interface command. If a Human Interface job contains objects whose disabling depths are greater than one, the operator cannot cancel the command by entering CONTROL-C. For most commands, the ability to cancel via CONTROL-C is desirable, if not required. To prevent other similar situations, it is recommended that you use DISABLE$DELETION, ENABLE$DELETION, and FORCE$DELETE only in 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 and when disabling deletion.

10.9 SYSTEM CALLS FOR EXTENDING THE OPERATING SYSTEM

The following system calls are used extensively by OS extensions:

DISABLE$DELETION--increases the deletion disabling depth of an object by one.

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

FORCE$DELETE--deletes objects whose disabling depths are one or zero.

RQE$SET$OS$EXTENSION--attaches the entry point address of the OS extension to a call-gate. Users familiar with the iRMX 86 Operating System should be aware that this system call replaces SET$OS$EXTENSION. When writing an OS extension, you must use this system call.

SIGNAL$EXCEPTION--advises a task that an exceptional condition has occurred in an OS extension that the task has called.

For a complete list and explanation of the iRMX II Nucleus system calls, see the *iRMX II Nucleus System Calls Reference Manual.*

Intel®

CHAPTER 11 TYPE MANAGERS

11.1 INTRODUCTION

The object types and system calls provided by the Nucleus and I/O System are sufficient for many applications. However, some applications require additional object types and system calls for manipulating them. 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 a new type. The end of this chapter contains an example that extends the operating system and creates and deletes objects of a new type.

11.2 CREATING NEW OBJECTS

Creating customized 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 will compose the new object (the <u>component objects</u>) and returns a token for the new object, called a <u>composite object</u>.

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

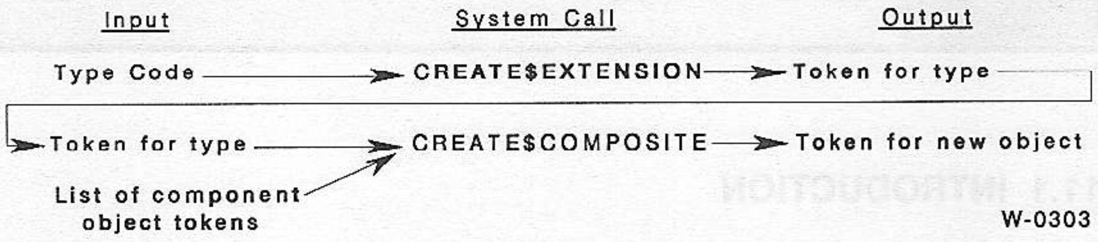


Figure 11-1. Creation Sequence for Composite Objects

Note these two facts when creating a composite object:

1. Its components, called component objects, are all iRMX II objects, either Intel- or user-provided.
2. No structure is imposed on composite objects of a given extension type. Two objects of the same extension type can be 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. Only the type manager 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.)

11.3 MANIPULATING COMPOSITE OBJECTS AND EXTENSION TYPES

Two system calls manipulate 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, with either another token or a null.

11.4 DELETING COMPOSITE OBJECTS AND EXTENSION TYPES

Two system calls delete 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 all 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 (it returns an exception code if the job contains any extension objects), it can delete composites or send them to deletion mailboxes where their type managers delete them.

The deletion$mailbox parameter in the CREATE$EXTENSION system call determines whether DELETE$EXTENSION and DELETE$JOB delete composite objects or send them to deletion mailboxes. This parameter has two options:

If you specify SELECTOR$OF(NIL) for the parameter, then DELETE$EXTENSION and DELETE$JOB assume all responsibility for deleting composite objects. The type manager plays no part in the deletion process. In this case, you can skip the next three sections of this chapter.

If you specify a mailbox token for the parameter, then DELETE$EXTENSION and DELETE$JOB send tokens for all composite objects of the indicated type to the mailbox. The type manager is then responsible for deleting the composite objects.

Two conditions must be met before the type manager receives tokens for composite objects via the deletion mailbox:

The type manager, when it calls CREATE$EXTENSION, must fill in the deletion$mailbox parameter with a token for a mailbox.

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

The following sections describe the type manager's responsibilities in more detail.

11.4.1 Type Manager Responsibilities During DELETE$JOB

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

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

1. It must wait at the deletion mailbox to receive the tokens for the objects to be deleted.

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

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

4. It should call DELETE$COMPOSITE, which deletes the composite object (but not the component objects) and informs the Nucleus that the type manager has finished the special processing that deletes the composite object. After the type manager calls DELETE$COMPOSITE, the Nucleus resumes the DELETE$JOB processing.

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

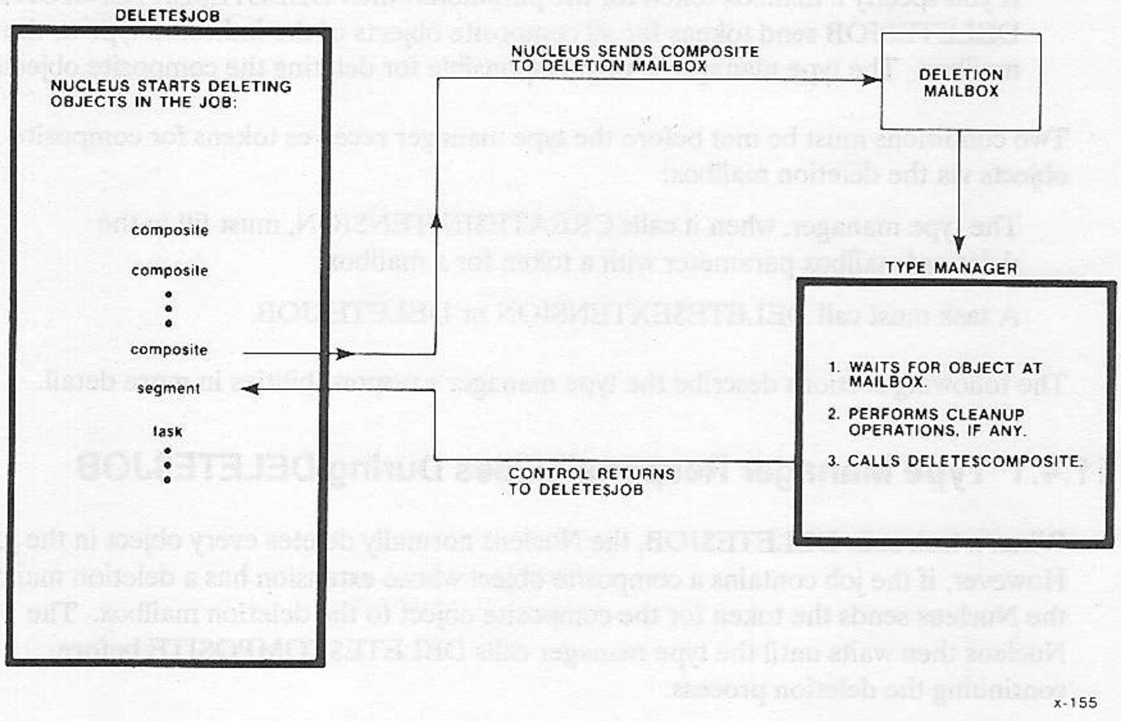

DELETE$JOB
NUCLEUS STARTS DELETING OBJECTS IN THE JOB:
composite
composite
composite
segment
task
NUCLEUS SENDS COMPOSITE TO DELETION MAILBOX
DELETION MAILBOX
TYPE MANAGER
1. WAITS FOR OBJECT AT MAILBOX.
2. PERFORMS CLEANUP OPERATIONS, IF ANY.
3. CALLS DELETE$COMPOSITE
CONTROL RETURNS TO DELETE$JOB
x-155

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

Note that the type manager is not required to delete all component objects. In the course of DELETE$JOB, the Nucleus deletes any Nucleus objects in the job. The Nucleus sends the tokens for any I/O System, Extended I/O System, or Human Interface (all three are OS extensions) objects to their respective deletion mailboxes, where the subsystems themselves delete the objects. The Nucleus sends the tokens for 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.

11.4.2 Type Manager Responsibilities During DELETE$EXTENSION

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

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

However, unlike DELETE$JOB processing, the type manager has the choice during DELETE$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$JOB, DELETE$EXTENSION does not delete any component objects.

11.4.3 Deletion of Nested Composites

Since a composite object can contain objects of any kind, some of its component objects may be composite objects themselves. This 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 to complete its processing.

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

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

DELETE$JOB deletes composite objects before it deletes non-composite objects. It deletes composite objects on a last-in-first-out basis; that is, in the reverse order from which they were created. Therefore, a type manager can depend on receiving the tokens for composite objects that it creates before the Nucleus deletes the component objects contained in them. The only exception is when a composite (composite A) is created before another composite (composite B), and composite B is inserted as a component into composite A using ALTER$COMPOSITE. 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.

11.5 WRITING A TYPE MANAGER

A type manager consists of two parts:

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

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

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

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

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.

11.6 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 should be imitated with discretion. In particular, the example has the following shortcomings:

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

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

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

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

The example has been desk-checked, but the example has not actually been tested.

A ring buffer is a block of memory in which bytes of data are placed at successively higher addresses. Interspersed with byte insertions are byte removals, with the restriction that the byte being removed must always be the byte that has been in the buffer for the longest time. Thus, data enters and leaves a ring buffer in a FIFO manner. Ring buffers are so named because 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 containing data "runs" around the ring, with the pointer to the last byte out "chasing" the pointer to the last byte in. Figure 11-3 illustrates these characteristics.

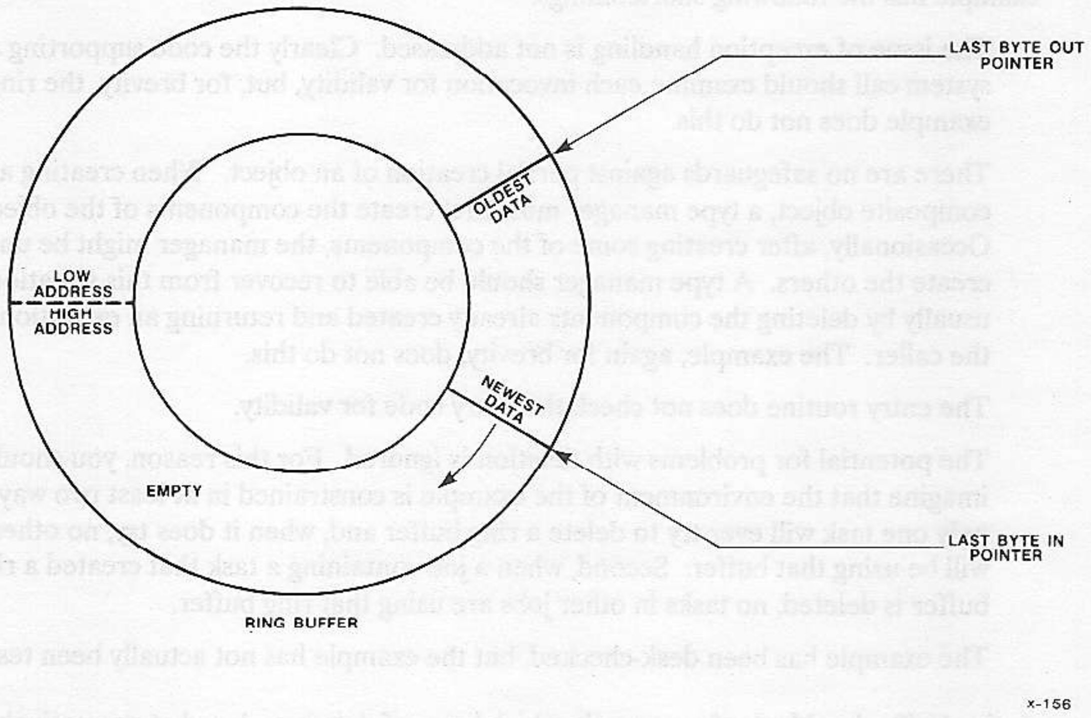


Figure 11-3. A Ring Buffer

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

```
/***********************************************************************
 *  NOTE:  The following common literal file (COMMON.LIT) is included   *
 *  in each of the PL/M-286 portions of the example.                     *
 ***********************************************************************/

   DECLARE TOKEN              LITERALLY 'SELECTOR';
   DECLARE forever        LITERALLY 'WHILE 1';
   DECLARE indefinitely             LITERALLY 'OFFFFH';
   DECLARE ASTR$STRUC         LITERALLY 'STRUCTURE(
                          num$slots       WORD,
                          num$components  WORD,
                          seg             TOKEN,
                          empty$ct        TOKEN,
                          full$ct         TOKEN)';
   DECLARE POINTER$STRUC      LITERALLY 'STRUCTURE(
                          offset          WORD,
                          selector        SELECTOR)';

   DECLARE SEGMENT$STRUC      LITERALLY 'STRUCTURE(
                          size            WORD,
                          head            WORD,
                          tail            WORD,
                          buffer(1)       BYTE)';
```

11.6.1 Initialization

The initialization creates a region to protect data in ring buffers from being manipulated by more than one task at a time. This part of the OS extension also creates the required extension type, creates a deletion mailbox, and then waits at the deletion mailbox. The OS extension call-gates are established during configuration. For this example, they are GDT slots 440H, 441H, 442H and 443H. Code for the initialization includes the following:

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

```
   RING$BUFFER$MANAGER:  PROCEDURE EXTERNAL;
   END RING$BUFFER$MANAGER;

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

   ring$buffer$region = RQ$CREATE$REGION (
       fifo,
       @exception);
   deletion$mbox = RQ$CREATE$MAILBOX (
       fifo,
       @exception);
   ring$buffer$type=RQ$CREATE$EXTENSION (
       rb$code,
       deletion$mbox,
       @exception);
   CALL RQ$END$INIT$TASK;
   DO FOREVER;
       delete$object = RQ$RECEIVE$MESSAGE (
           deletion$mbox,
           indefinitely,
           @response$mbox
           @exception);
 /***************************************************************************
  * If desired, delete the components of the composite object.  They are *
  * not automatically deleted when DELETE$EXTENSION is called.  See the  *
  * DELETE$RING$BUFFER procedure, shown later, for the code that does    *
  * this.                                                                 *
  ***************************************************************************/

       CALL RQ$DELETE$COMPOSITE (
           delete$object,
           @exception);
       END;    /* FOREVER */
   END RING_BUFFER_INIT;
```

11.6.2 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-286 LARGE model. If a different model had been used for compiling the application code, these interface procedures would be slightly different, reflecting the fact that, when making procedure calls in other models, the stack is used differently than in the LARGE model. The interface procedures are as follows:

```
CREATERB    PROC     FAR
            PUBLIC   CREATERB
            EXTRN    GATE_440: FAR

            PUSH     BP
            MOV      BP,SP
            PUSH     BP+6      ; parameter--the size of the ring buffer
buffer      CALL     GATE_440      ; call the OS-extension via a call-gate
            POP      BP
            RET      2
CREATERB    ENDP

DELETERB    PROC     FAR
            PUBLIC   DELETERB
            EXTRN    GATE_441: FAR

            PUSH     BP
            MOV      BP,SP
            PUSH     BP+6      ; parameter--the ring buffer to delete
            CALL     GATE_441      ; call the OS-extension via a call gate
            POP      BP
            RET      2
DELETERB    ENDP

GETRBYTE    PROC     FAR
            PUBLIC   GETRBYTE
            EXTRN    GATE_442: FAR

            PUSH     BP
            MOV      BP,SP
            PUSH     BP+6      ; parameter--ring buffer to read from
            CALL     GATE_442      ; call the OS-extension via a call-gate
            POP      BP
            RET      2
GETRBYTE    ENDP
PUTRBYTE    PROC     FAR
            PUBLIC   PUTRBYTE
            EXTRN    GATE_443: FAR
            PUSH    BP
            MOV     BP,SP
            PUSH    BP+8          ; parameter--the character to write
            PUSH    BP+6          ; parameter--ring buffer to write to
            CALL    GATE_443      ; call OS-extension via a call-gate
            POP     BP
            RET     4
PUTRBYTE    ENDP
```

These interface procedures correspond to a set of external procedure declarations in the application PL/M-286 code:

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

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

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

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

11.6.3 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 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 no components will be added, 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. The third word points to the oldest byte in the buffer. The remainder of the segment is 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, then creates the composite object, as follows:

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

CREATE_RING_BUFFER:
    PROCEDURE (size) TOKEN PUBLIC REENTRANT;
    DECLARE size               WORD;
    DECLARE seg$ptr            POINTER;
    DECLARE ptr$struc          POINTER$STRUC AT (@seg$ptr);
    DECLARE astr                   ASTR$STRUC;
    DECLARE segment            SEGMENT$STRUC BASED seg$ptr;
    DECLARE exception          WORD;
    DECLARE ring$buffer        TOKEN;
    DECLARE priority               LITERALLY '1';

    astr.num$slots = 3;
    astr.num$components = 3;
    astr.seg = RQ$CREATE$SEGMENT (
        size+6,
        @exception);
    astr.empty$ct = RQ$CREATE$SEMAPHORE (
        size,
        size,
        priority,
        @exception);
    astr.full$ct = RQ$CREATE$SEMAPHORE (
        0,
        size,
        priority,
        @exception);
    ptr$struc.base = astr.seg;
    ptr$struc.offset = 0;
    segment.size = size;
    segment.head = -1;
    segment.tail = 0;
    ring$buffer = RQ$CREATE$COMPOSITE (
        ring$buffer$type,
        @astr,
        @exception);
    RETURN ring$buffer;
END CREATE_RING_BUFFER;
```

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

11.6.4 The Delete Ring Buffer Procedure

DELETE_RING_BUFFER, which can be called by any task, deletes a ring buffer.

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

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

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

11.6.5 The Put Byte Procedure

PUT_BYTE places a character in the buffer by advancing the pointer to the front of the buffer then placing the character in the byte being pointed to.

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

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

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

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

11.6.6 The Get Byte Procedure

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

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

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

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

11.6.7 Epilogue

This completes the important parts of the example (recall that the example is not complete). Any task in any job linked to these procedures may call any one of the procedures. The procedure names to be used in such calls are CREATE$RB, DELETE$RB, GET$RB$BYTE, and PUT$RB$BYTE. Note that application programs cannot manipulate either ring buffers or their component objects, except through these system calls. In fact, application programmers need not 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 II objects.

11.7 SYSTEM CALLS FOR TYPE MANAGERS

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

ALTER$COMPOSITE--replaces a component in a composite object with either a null or another object.

CREATE$COMPOSITE--creates a composite object of a specified extension type.

CREATE$EXTENSION--creates an extension object that may subsequently be used as a license for creating composite objects. Input may include a token for a mailbox where these composite objects are sent for deletion.

DELETE$COMPOSITE--deletes a composite object.

DELETE$EXTENSION--deletes an extension object and optionally, sends all composite objects of that extension type to the associated deletion mailbox.

INSPECT$COMPOSITE--returns a list of the component object tokens contained in a composite object.

For a complete list of the iRMX II Nucleus system calls, see the *iRMX II Nucleus System Calls Reference Manual.*

Intel®

CHAPTER 12 iRMX® II MULTIBUS® II SYSTEMS

12.1 INTRODUCTION

This chapter is intended for programmers who need to understand the basic concepts of the MULTIBUS II architecture as it relates to the Extended iRMX II.3 Operating System. The following topics are discussed:

- An analogy of how MULTIBUS II works
- An overview of MULTIBUS II hardware
- An overview of Extended iRMX II Nucleus Communication Service
- Examples of using the Extended iRMX II MULTIBUS II services
- A glossary of terms used in this chapter

For a list of documents that contain more detailed discussions of the MULTIBUS II bus architecture, see the Related Publications section in the Preface of this book.

Throughout this chapter, the term "agent" is used interchangeably with "board" to mean any board that contains a Message Passing Coprocessor (MPC) and resides in one of the MULTIBUS II Parallel System Bus (iPSB) slots.

12.2 FEATURES OF MULTIBUS® II SYSTEMS

Extended iRMX II and MULTIBUS II together form an easy-to-use and reliable computer system. Among the features provided by this architecture are:

- High-performance 32-bit-wide reliable bus with a 40M bytes per second transfer rate
- Multiprocessor support
- Interprocessor communication via message passing
- System calls that free the user from needing to know details of how data is sent using MULTIBUS II message passing
- 255 virtual interrupts per board
- Geographic addressing of bus agents by cardslot number (ID)

12.3 AN ANALOGY OF HOW MULTIBUS® II SYSTEMS WORK

The MULTIBUS II bus architecture defines a connectionless mode of data transfer. In this type of data transfer, the information is sent in a format called a message or datagram. The principle of sending these messages is similar to sending a letter through the mail. You write a letter, put it in an envelope and address it to the person that you want to receive it. If you want to be certain that the person answers your letter, you put an RSVP and your return address in the letter. You drop your letter into a mailbox and wait. The letter is delivered and answered using the information you provided. This is connectionless because you do not check to see if the person in question still lives at the same address before you send your letter. If the letter is not deliverable, it will be returned to you.

Systems based on the MULTIBUS II architecture perform data transfers between boards in this connectionless message format. All the information needed to direct the data (message) to its destination, explain what is wanted, and identify where to send the answer, is included in the message. Both the system's hardware and operating system perform functions in directing and moving this data.

In traditional architectures (ones that rely on connections), when processor or controller boards need to communicate they use shared memory. This shared memory is set aside during the system configuration and is available as "wake-up" addresses for both kinds of devices. This type of data transfer is similar to making a telephone call. First you dial the number, wait for an answer on the other end of the line, and then have your conversation. While you are on the phone, neither party can make or accept any other phone calls.

In the telephone analogy above, the phone line represents the system bus of a computer. Whenever any two processors are involved in communication, the system throughput is limited to the data transfer rate of the slower device. MULTIBUS II systems overcome this and other limitations of the shared memory approach.

The rest of this chapter divides the explanation of iRMX II and MULTIBUS II into first a hardware overview, followed by a software overview.

12.4 MULTIBUS® II HARDWARE OVERVIEW

The MULTIBUS II bus architecture consists of six buses, as shown in Figure 12-1:

- <u>The Parallel System Bus (iPSB)</u> is a high-performance, general-purpose, 32-bit bus that provides system data movement and interprocessor communication facilities. It can be thought of as the "message passing" bus.
- <u>The Local Bus Extension (iLBX II)</u> is an extension of the on-board processor bus that provides arbitration-free, high-bandwidth access to local memory.
- <u>The Serial System Bus (iSSB)</u> is a low-cost, one-bit alternative to the iPSB bus that adds flexibility to meet the requirements of a wide range of systems.
- <u>The MULTICHANNEL DMA I/O Bus</u> is retained from the MULTIBUS I bus architecture. It allows high-speed block transfers of data over the shared data path between custom peripherals and single board computers.
- <u>The iSBX I/O Expansion Bus</u> is retained from the MULTIBUS I bus architecture. It allows incremental board expansion through the addition of small iSBX MULTIMODULE boards.
- The <u>BITBUS Interconnect</u> is a serial bus, optimized for the high-speed transfer of short control messages and implemented as a pair of twisted wires.

In addition to the six buses, the MULTIBUS II bus architecture consists of four separate address spaces:

- <u>Message address space</u> is the range of addresses that identify all iPSB agents that send and receive messages. Each agent is assigned a one-byte message address that uniquely identifies that board in the system. The term "host ID" is used interchangeably with the term message address.
- <u>Interconnect address space</u> is a set of 512 byte-wide registers that provide identification and configuration information for each message-passing agent. Each board contains its own set of interconnect registers.
- <u>I/O address space</u> is the I/O port address range that serves as a system-wide interface to terminal controllers, mass storage devices, and other peripheral devices.
- <u>Memory address space</u> is the address range for storing and retrieving data and code. A MULTIBUS II system can have up to 4 gigabytes of memory. Each Extended iRMX II agent can have up to 16M bytes of memory for its own use.

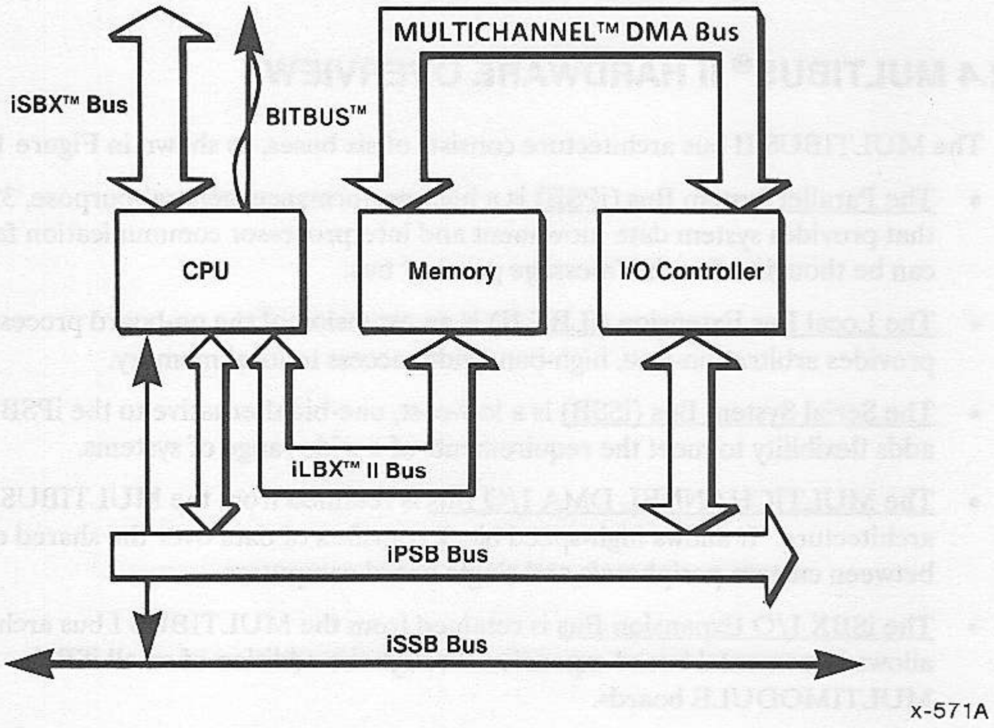


Figure 12-1. Simplified MULTIBUS® II Bus Architecture

12.4.1 Central Services Module (CSM)

The iSBC CSM/001 Central Services Module, or CSM, is an agent that coordinates certain system-level services and functions common to all agents. The CSM board must be present in every MULTIBUS II system and must be installed in cardslot 0 of the backplane. The CSM board provides these system services:

- It generates, on power-up or cold reset, the message addresses (cardslot IDs) and arbitration numbers (arbitration IDs) for all agents connected to the iPSB bus.
- It provides a central source for the iPSB clock signals (BCLK* and CCLK*).
- It generates system wide reset signals on power-up (RST*), cold reset (CRST*), or warm reset (WRST*).
- It monitors timeout error conditions and generates the timeout error (TIMOUT*).
- It maintains the system wide, battery backed-up, global time-of-day clock.

* A hardware signal that is active when low (0 volts.)

12.4.1.1 Global Time-of-Day Clock

The CSM board maintains an on-board, battery backed-up global time-of-day clock. This clock is used by every agent in the system that requires a clock. Agents in the system can maintain a local time-of-day clock that is a copy of the global clock. Both the global and local clocks keep track of:

- The current day (day, month, and year)
- The current time (hours, minutes, and seconds)

The two types of clocks are necessary because accessing the global clock takes much longer than accessing a local clock. Each agent with a local clock accesses the global clock only on system reset or at the request of the user.

Two Basic I/O system calls allow your application to read or set the global clock. See the *Extended iRMX II Basic I/O System Calls Reference Manual* for information on these calls.

Two Extended iRMX II Human Interface commands (DATE and TIME) can be used to read or set the global clock from a terminal (for the super user only.) See the *Operator's Guide To The Extended iRMX II Human Interface* for information on these commands.

12.4.2 Interconnect Address Space

On each MULTIBUS II agent is an area called the interconnect space or interconnect registers. This space is a set of 512 8-bit registers used for dynamic software-controlled initialization, configuration, testing, and error diagnosis. Part of their function can be described as electronic jumpers. With a MULTIBUS II board the board configuration is changed by writing values into interconnect registers, not by installing or removing physical jumper connections. Each interconnect register is addressable by its own 16-bit interconnect address (interconnect ID.) The interconnect ID consists of the cardslot ID of the agent (0 through 19 for iPSB cardslots, 24 through 29 for iLBX II cardslots, and 31 for the host processor board) and the interconnect register number (0 through 511).

Interconnect registers 0 and 1 contain the Intel-assigned vendor ID for the vendor of the particular board. The vendor ID is read-only; write operations to these two registers are ignored. Interconnect registers 2 through 511 contain such board specific attributes as board ID, revision number, cardslot ID, type of board (e.g. processor, I/O controller), and iPSB starting and ending addresses. The contents of these registers are board-dependent. Register values are read and modified through the interconnect address space. Refer to the board's hardware reference manual for details on interconnect register usage.

During power-up or cold reset and through each board's interconnect address space, the Bootstrap Loader, the Root Job, and the system monitor can automatically configure the boards in the system to the configuration you choose. During power-up the system monitor can initialize any read/write interconnect registers. Two system calls are provided that allow programs to dynamically read (get) or write (set) the contents of any interconnect register on any board in the system. See the *Extended iRMX II Nucleus System Calls Reference Manual* for information on these system calls.

12.4.3 Built-In Self Tests (BIST)

Each board in a MULTIBUS II system contains a set of firmware-based diagnostic tests that, on power-up, does some internal checking and assigns a "go" or "no-go" condition to the board based on the results of the test. These tests are called the Built-In Self Tests or BIST.

On power-up, or cold reset, each board's BIST automatically invokes its initialization checks and diagnostic tests. If successful, the BIST initializes the board to a predefined state and clears its RSTNC* (reset-not-complete) signal. However, if a test fails, the BIST assesses a "no-go" condition, flags the error, and ceases initialization of that board. After an error, the BIST Test ID register (in the board's interconnect space) contains the number of the test that failed, so the problem can be identified and corrected.

12.4.4 The MULTIBUS® II Message Passing Hardware and Message

The entire MULTIBUS II architecture of six buses and four types of address space was designed to perform the function of sending data between agents through one of the local buses in a traditional manner, or over one of the message passing buses in the form of packets of information called messages. A MULTIBUS II message is a variable length sequence of bytes (called a packet) that provides a means for one bus agent to communicate with another. All the information needed to know which agent sent the message and what the sending agent wants is stored in the first "packet."

Each agent on the bus has a Message Passing Coprocessor (MPC) chip that performs the message passing functions. When a message comes across the bus, each agent checks a portion of the message header that contains the message address of the destination agent. It compares the destination address with its own agent ID, if they match it retrieves the entire message packet from the bus. If the addresses do not match the message is ignored.

When a message is retrieved from the bus, more bytes are examined to determine:

- The agent ID of the sending board.
- Is this message a reply to a previous message or is it a new communication being started by another agent.

- Which port on the board is to receive the message?
- Is a response necessary?
- Are more "packets" coming that are part of the same message?

Figure 12-2 shows a simplified MULTIBUS II message packet. Although the specific format of messages is different for different types of messages, blocks of bytes can be identified by what part of the entire system reads and uses those bytes.

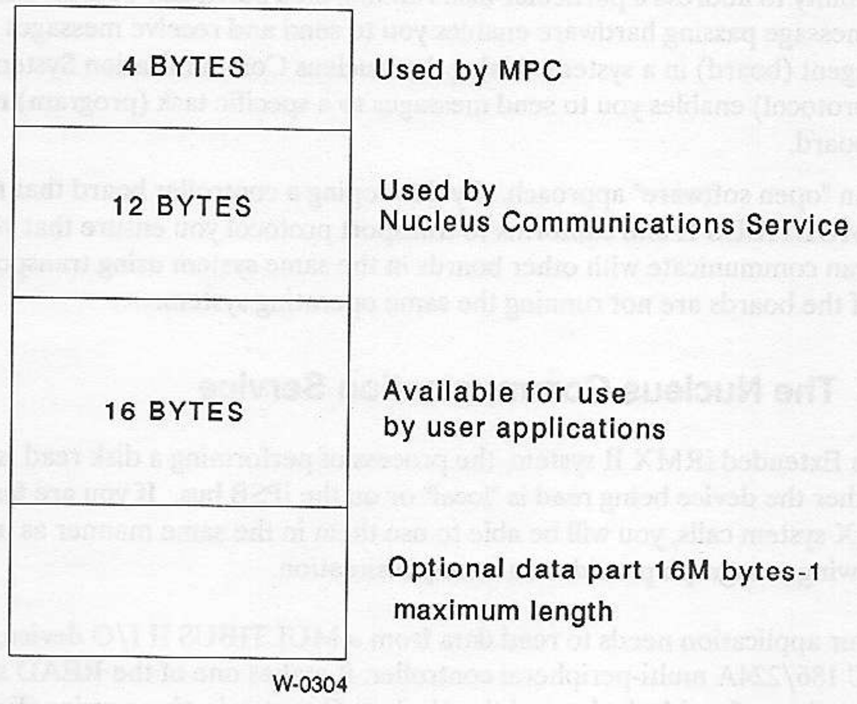

4 BYTES
Used by MPC
12 BYTES
Used by Nucleus Communications Service
16 BYTES
Available for use by user applications
Optional data part 16M bytes-1 maximum length
W-0304

Figure 12-2. A Simplified MULTIBUS® II Message

12.5 EXTENDED iRMX® II SOFTWARE OVERVIEW

The Extended iRMX II Operating System is largely concerned with managing the resources available to a single processor board. These resources are I/O devices and RAM boards that are related to a processor board so that the the resources appear to be physically on the processor board. For example, RAM boards are typically placed on the Local Bus Extension (iLBX) bus. Boards on this bus appear to the processor to be a part of its own board resources.

12.5.1 The MULTIBUS® II Transport Protocol

Extended iRMX II also supports the concept of gaining access to resources and services on boards distributed on the MULTIBUS II Parallel System Bus (iPSB) bus through a new feature called the Nucleus Communications Service, which is an implementation of Transport Protocol. The Transport protocol is a software interface that works with the MULTIBUS II message passing hardware. The Nucleus Communication layer is the Extended iRMX II implementation of the MULTIBUS II Transport Protocol. Among other things it provides:

- an interface that hides many of the details of sending data (messages) over the message passing bus, the Parallel System Bus (iPSB).
- ability to address a particular task running on a particular board. Using just the message passing hardware enables you to send and receive messages to a specific agent (board) in a system. Using the Nucleus Communication System (transport protocol) enables you to send messages to a specific task (program) running on a board.
- an "open software" approach. By developing a controller board that runs on MULTIBUS II and conforms to transport protocol you ensure that your controller can communicate with other boards in the same system using transport protocol, even if the boards are not running the same operating system.

12.5.2 The Nucleus Communication Service

In an Extended iRMX II system, the process of performing a disk read is largely the same whether the device being read is "local" or on the iPSB bus. If you are familiar with the iRMX system calls, you will be able to use them in the same manner as in the past. The following paragraph provides an example situation.

If your application needs to read data from a MULTIBUS II I/O device, such as the iSBC 186/224A multi-peripheral controller, it makes one of the READ system calls. The device driver for this device and the Nucleus Communication service (Extended iRMX II implementation of Transport Protocol) translate this read into a MULTIBUS II message. The I/O device's controller board receives this message and sends back a response message to inform your processor that it either can or cannot perform the read. If the I/O controller can fill the request, a series of messages is sent between the two agents (boards) defining how the actual read will be performed. When this short series of messages is complete the actual data read is performed and the data is sent to your application as another group of messages. When the entire process is complete, the MPC on the board where your application is running sends an interrupt to the CPU informing it of the completion of the data read. The entire message passing process is transparent to your application.

12.5.3 Nucleus Communication Objects

Two objects and system calls to manipulate them have been added to the Operating System to provide an interface to the MULTIBUS II hardware. The following sections explain the port and the buffer pool, the new objects.

12.5.3.1 Port

A port is a bi-directional communications access point for tasks running on different agents. Ports provide a level of addressing that permits sending data to a particular task (program) running on a board. The port is the software interface to the message passing hardware. All of the "send" and "receive" message system calls have parameters that identify the ports that are involved in the operation.

You create a port with the RQ$CREATE$PORT system call. Two types of ports can be created, a data port that is used to send and receive relatively large amounts of data, and a signal port that is used to send and receive control signals only. The signal port is provided for compatibility with systems that use the Message Interrupt Controller (MIC).The MIC is a subset of the MPC available on earlier products.

When you create a port you specify the following information in the system call:

- The number of simultaneous transactions that can be outstanding at the port. (Ignored for signal ports.)
- What type of messages this port can send and receive.
- Whether tasks waiting for messages will be queued in FIFO or priority order.
- A port ID, which is a number that uniquely identifies this port on this board. The user can enter this, or enter a zero which tells the Nucleus Communication Service to create the port ID.
- What the system should do if an outgoing message is too large to fit in any one buffer available at the destination port. You can specify that the message can or cannot be broken into pieces if it is too large to fit in a single buffer. If you specify that messages cannot be broken into pieces, an error will be returned if the situation occurs.
- As with all system calls, you specify where the condition code generated from the call should be placed.

Ports are deleted from the system using the RQ$DELETE$PORT system call.

System calls have been added to provide flexibility in manipulating ports. The following paragraphs discuss these system calls.

The RQ$CONNECT system call provides a method of specifying a default destination for messages sent from a port. When a port issues this call, it is logically connected to another remote port. After issuing the RQ$CONNECT call, no remote destination (socket, a two-word data structure shown below) is specified when sending messages, because the connected port (default) is assumed.

```
socket        LITERALLY 'STRUCTURE(
                                    host_id      WORD,
                                    port_id      WORD)';
```

When receiving messages, any message that does not have this default port socket as its source is ignored. This call affects only the port that issues it, not the port that is connected. A remote port that is connected is not limited to sending and receiving messages to the connecting port.

The RQ$ATTACH$PORT system call permits the forwarding of messages. A port may specify that all messages sent to it be forwarded to another port. In this arrangement, the port that is forwarding its messages is referred to as the source port and the port that receives the messages is referred to as the sink port. One level of forwarding is permitted, that is a sink port may not forward its messages. The message forwarding is canceled by issuing an RQ$DETACH$PORT system call.

The RQ$ATTACH$BUFFER$POOL system call provides a port with memory resources called buffer pools (discussed in a later section.) These memory resources are used in receiving large data messages. These memory resources can be detached from the port by using the RQ$DETACH$BUFFER$POOL system call.

The RQ$GET$PORT$ATTRIBUTES system call allows a task to get information about any port. The information returned is the same information discussed above as being specified when a port is created, plus the following: does the port have a default socket and is it forwarding its messages.

12.5.3.2 Buffer Pools

Buffer Pools are holding areas for segments which in MULTIBUS II systems are usually associated with a port. Having a pool of memory readily available to a port cuts down on system overhead because allocating the existing buffers is faster than creating and deleting segments.

Buffer pools are empty when created. The user gives segments to the buffer pool. The segments are created using the the RQ$CREATE$SEGMENT system call. The created segments are given to a buffer pool by using the RQ$RELEASE$BUFFER system call. The buffers are then used by tasks that require memory. Both MULTIBUS I and MULTIBUS II systems can use buffer pools. In MULTIBUS II systems, ports require an associated buffer pool as a holding area for messages. Any task that requires frequent creation and deletion of segments may improve performance by using a buffer pool with pre-allocated segments.

Buffer pools incur a certain amount of system overhead in their creation. The following formula defines the amount of resources required.

(Max Buffers * 4) + 108 bytes = the amount of memory used by any given buffer pool.

When you create a buffer pool you specify the following information:

- The maximum number of buffers that can reside in the buffer pool at any one time (8192 maximum.)

- Whether or not the buffer pool supports data chains. Data chains are a method of receiving messages that are larger than any single buffer can hold. Note that data chaining is one of two supported methods for receiving messages larger than any single buffer. The other method is message fragmentation.

12.5.3.2.1 System Calls for Buffer Pools

The RQ$CREATE$BUFFER$POOL system call creates a buffer pool object. Each buffer pool object acts as a holding area for segments (buffers.) When a buffer pool is created it is not associated with any port, see RQ$ATTACH$BUFFER$POOL in an earlier section. Buffer pools are deleted using the RQ$DELETE$BUFFER$POOL system call.

The RQ$REQUEST$BUFFER system call gets a buffer from the specified buffer pool. Ideally, the data will fit into an existing buffer; if this is not possible, a special method called data chaining is used. Buffers are returned to the buffer pool using the RQ$RELEASE$BUFFER system call.

12.5.3.2.2 Data Chains

When a buffer pool is created, you can set a bit in the flag field that specifies support for data chains. Data chaining is supported only on processor boards that contain the ADMA (Advanced Direct Memory Access) device.

In buffer pools that support data chaining, if a message is too large to fit in any single buffer, the message can be broken into smaller pieces. These pieces are placed in smaller non-contiguous buffers and a data structure called a data chain block is used to keep track of the different parts of the message. Data chains are created automatically by the receiving board's hardware, but the user must extract the data by using the information in the data chain block. No element of a data chain can be longer than 64K-2 bytes. Figure 12-3 illustrates a data chain block and a data chain.

Data chains incur a certain amount of system overhead in their creation. The minimum data chain block size can be computed by:

(Max_elements * 8) + 2 bytes
Where:

Max_elements	is a configuration option Maximum Data Chain Elements (MCE) from the ICU Nucleus screen.

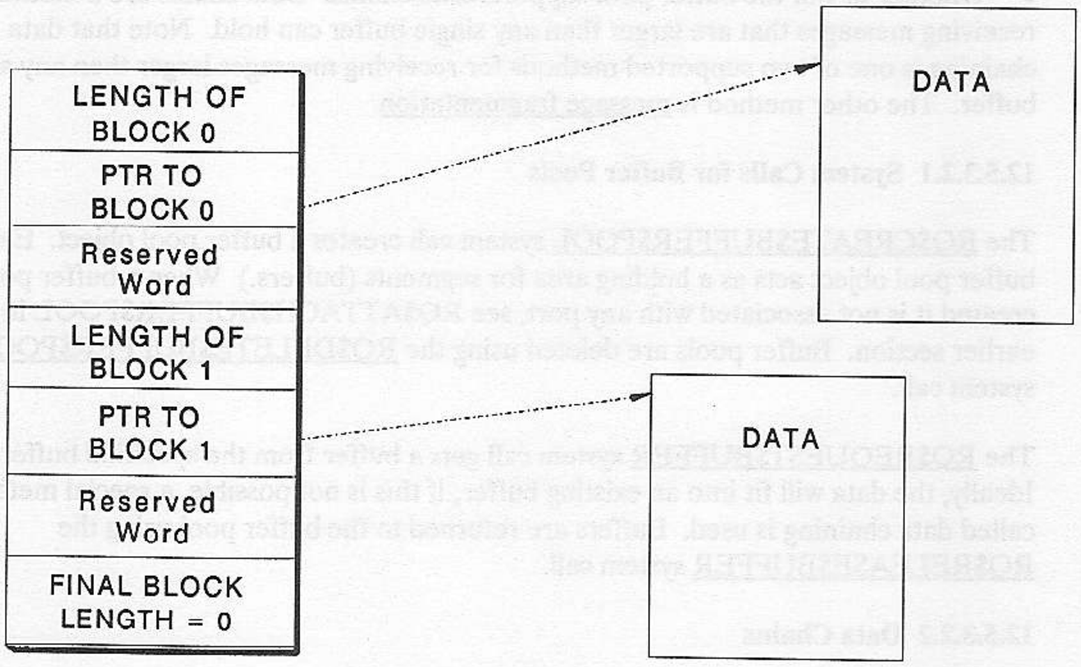


Figure 12-3. A Data Chain Block and a Data Chain W-0307

12.5.3.2.3 Message Fragmentation

When setting up a data port you can set a bit in the flag field that enables or disables a method of breaking large messages into smaller pieces when no single buffer is large enough for the message. For the port object, this process is called message fragmentation. Messages can be fragmented by the sending board, send fragmentation or fragmented by the receiving board, receive fragmentation.

Message fragmentation is performed as a series of messages sent between the agent sending the data and the agent receiving the data. Because multiple messages must be sent and received to perform message fragmentation, there is more system overhead involved in message fragmentation than in data chaining.

12.5.4 System Calls to Work With MULTIBUS® II Message Space

In addition to the system calls that manipulate the MULTIBUS II objects, system calls that provide an interface to the four types of address space, discussed in the hardware overview, are provided. A complete interface to the MULTIBUS II bus architecture is provided by these system calls.

12.5.4.1 System Calls for Interconnect Space

The RQ$GET$INTERCONNECT system call reads the contents of one byte-wide interconnect register on one board for each invocation of the call. This call is used for, among other things, finding out how many and what type of boards are in a system, and what the HOST$ID (message address of a board) is.

The RQ$SET$INTERCONNECT system call writes over the contents of one byte-wide interconnect register on one board for each invocation of the call. (Some interconnect registers are read only.) One use for this call is to dynamically reconfigure a board.

12.5.4.2 System Calls for Sending Messages through Message Space

The RQ$SEND system call sends a message to a remote host without any request for a response. The message can be a control message only or can be a control message and contain a POINTER to a data portion of the message.

The RQ$SEND$RSVP system call initiates a message interchange that is used to send large amounts of data from one board (host) to another. This call specifies that a reply is requested.

The RQ$SEND$REPLY system call is used to answer a previous RQ$SEND$RSVP system call.

The RQ$BROADCAST system call is used to send a control message to each board in the system. Note that only one port on each board can receive this message.

The RQ$CANCEL system call cancels a message transmission. It is used to end data transfer initiated by an RQ$SEND$RSVP system call.

The RQ$RECEIVE system call initiates a message reception at a port. If the message contains a data portion, the receiving port must have already allocated a buffer large enough to hold the message before issuing the RQ$RECEIVE message. If no buffer is allocated, or no buffer is large enough, the message is rejected by the receiving agent.

The RQ$RECEIVE$REPLY system call accepts a reply to an earlier RQ$SEND$RSVP message. Sink ports, any port that receives forwarded messages, cannot issue this call.

The RQ$RECEIVE$FRAGMENT system call accepts a part of a message that was too large to fit in any buffer available at the receiving port. A buffer to receive the message fragment is specified in the call.

Message Space Calls that Support the MIC Device

The two calls are available that support the MIC device are:

The RQ$SEND$SIGNAL system call sends a signal message (dataless message) to a remote host (another board in the system.

The RQ$RECEIVE$SIGNAL system call picks up a signal message from the bus.

These are should be used only with systems that contain the MIC device.

12.5.4.3 Calls For Getting Information About Message Passing Agents (Boards)

The RQ$GET$HOST$ID system call, returns the HOST$ID, the message address of the calling task's board. It provides the host$id part of the host$id:port$id pair that makes up a socket. A socket is defined as a DWORD structure of the following format:

12.5.5 The Nucleus Communications Service System Calls

This section groups the Nucleus Communications Service system calls into functionally related groups.

12.5.5.1 System calls used with buffer pool objects

RQ$CREATE$BUFFER$POOL RQ$DELETE$BUFFER$POOL	Create or delete a buffer pool, a holding area for buffer segments. These buffer pools are used in conjunction with the port object as buffer space for messages sent or received at a port.

RQ$REQUEST$BUFFER RQ$RELEASE$BUFFER	Request or return a RAM buffer (segment) that is associated with a particular buffer pool. Buffers are created with the RQ$CREATE$SEGMENT system call.

12.5.5.2 System calls used with the port object

RQ$CREATE$PORT RQ$DELETE$PORT	Create or delete port objects. You specify the type of port, data or signal, in the RQ$CREATE$PORT system call.
RQ$CONNECT	Associate a port with a remote socket; such that, when a message is sent from that port it automatically goes to the connected port.
RQ$ATTACH$PORT RQ$DETACH$PORT	Create or remove a message-forwarding link between two ports. When a task issues the "ATTACH" call, the port that forwards its' messages is known as the <u>source</u> port. The port that is attached by the call, receives the forwarded messages, and is known as the <u>sink</u> port.
RQ$ATTACH$BUFFER$POOL RQ$DETACH$BUFFER$POOL	Attach or detach a pool of buffer segments to the specified port(s). This area of RAM is used as buffer space for messages sent to and from the specified port(s).
RQ$GET$PORT$ATTRIBUTES	Request information about a port. The following information is returned: the port type, maximum number of simultaneous transactions and discipline (FIFO or priority). Does it have a default socket, sink port, or buffer pool associated with it? Is message fragmentation supported?

12.5.5.3 System Calls Used to Send/Receive Messages Through Ports

RQ$BROADCAST
RQ$SEND
RQ$SEND$RSVP
RQ$SEND$REPLY
RQ$CANCEL
RQ$RECEIVE
RQ$RECEIVE$REPLY
RQ$RECEIVE$FRAGMENT

These calls are used to send and receive messages of the <u>data</u> transport protocol type. MULTIBUS II messages may consist of either:

<u>control-only</u>	<u>control + data</u>
32 bytes (unsolicited)	32 bytes plus up to 16 Mbytes-1 of data (solicited)

RQ$SEND$SIGNAL RQ$RECEIVE$SIGNAL	These calls are used to send and receive messages of the signal or <u>dataless</u> type. They should be used only in systems that contain the MIC device.

12.5.5.4 System Calls used with the Interconnect Registers on a board

RQGETINTERCONNECT RQ$SET$INTERCONNECT	These calls are used to read or write to the interconnect registers located on each MULTIBUS II host board.

12.5.6 Examples Using Nucleus Communications Service Calls

This section provides a conceptual explanation of most of the examples provided with the Operating System. The examples provide a more complete understanding of message passing as it relates to the Extended iRMX II Operating System. Each example includes a brief description of the operation of the example and all of the PL/M 286 code for the example.

All of the examples in this chapter are provided with the iRMX II Operating System. The MULTIBUS II examples discussed here are located in the path /RMX286/DEMO/PLM/MB2/INTRO. For a complete diagram of the iRMX II directory structure, see the *Operator's Guide to the Human Interface* manual. When you are ready to examine these examples, type:

```
ATTACHFILE /RMX286/DEMO/PLM/MB2/INTRO <CR>
```

to attach to the directory containing the example programs. To generate the executable modules for these examples type:

```
SUBMIT COMPILE <CR>
```

These two commands must be typed in on both host terminals, assuming that each host has its own disk.

Most of the examples use an external file called DCOM.EXT and a literal file called DCOM.LIT. Both of these files are presented at the end of this chapter.

The examples in this chapter are presented in an order similar to their use in a real system. The examples step you through the following concepts:

1. **Scanning the system** to determine what boards are in the system. This example runs independently of all the other modules.
2. **Creating a data transport protocol port** to use in message passing. This example runs independently of all the other modules.

3. **Sending an RSVP message** to another board and waiting for a reply. This module must be run with example 4 in this list or with example 7 in this list.
4. **Answering an RSVP message** from the receiving board. This module must be run with example 3 in this list.
5. **Sending a data chain message.** This example must be run with example 6 in this list.
6. **Receiving a data chain message.** This example must be run with example 5 in this list.
7. **Sending a fragmented message.** This example must be run with example 3 in this list.

The examples listed above make certain assumptions about the locations of the host boards in the MULTIBUS II system that they run on. The Figure 12-4 shows the required physical locations of the host boards (agents) in the system.

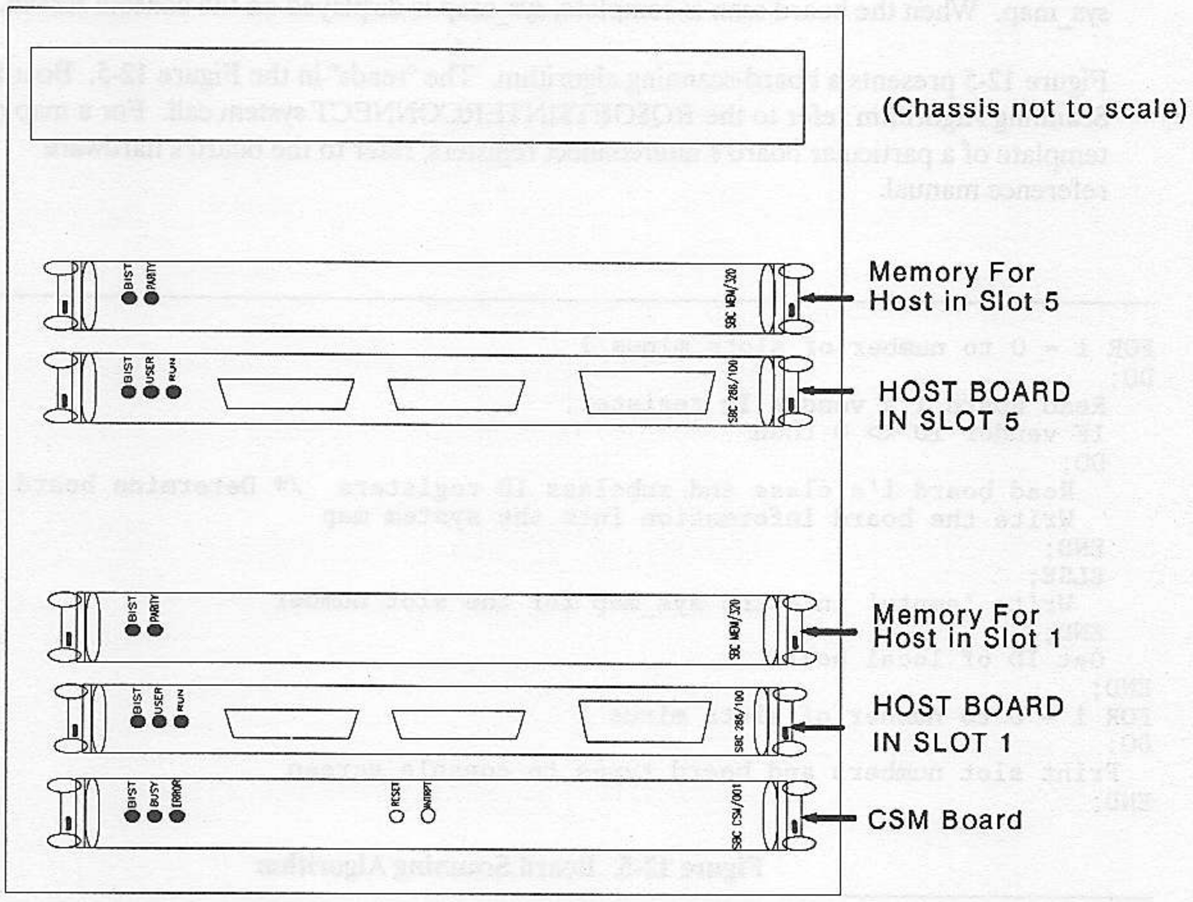


Figure 12-4. Physical Location of Boards in the Example

The MULTIBUS II examples directory also contains a larger example that is not shown in this chapter. This example implements a "name server", a program that permits the dynamic cataloging of all ports created in a system. A later section of this chapter discusses this example.

12.5.6.1 Interconnect Space Example

Before passing messages between agents (boards) in your system you need to determine what boards are in your system and the message addresses for the boards (host$id or cardslot number for boards on the iPSB bus.) You may also need to read or write the contents of a particular interconnect register. Writing a board scanner task allows you to dynamically determine host IDs, board type, and multiple occurrences (instances) of a board type.

This section presents an example of getting the interconnect information for an entire system. The example performs the board scan, getting the slot number and board type of each board in the system and places the information into an array of structures called sys_map. When the board scan is complete, sys_map is displayed on the console screen.

Figure 12-5 presents a board-scanning algorithm. The "reads" in the Figure 12-5. Board Scanning Algorithm refer to the RQ$GET$INTERCONNECT system call. For a map or template of a particular board's interconnect registers, refer to the board's hardware reference manual.

```
FOR i = 0 to number of slots minus 1
DO;
   Read board i's vendor ID register;
   IF vendor ID <> 0 then
   DO;
     Read board i's class and subclass ID registers  /* Determine board type */
     Write the board information into the system map
   END;
   ELSE;
     Write 'empty' into the sys_map for the slot number
   END;
   Get ID of local host
END;
FOR i = 0 to number of slots minus 1
DO:
  Print slot numbers and board types to console screen
END;
```

Figure 12-5. Board Scanning Algorithm

In the fourth line of the board scanner algorithm, a vendor ID of 0 (for iPSB hosts only) indicates that either the board was manufactured by a nonlicensed vendor or the cardslot is empty. If you are also scanning the iLBX II bus, replace the 0 with 0FFFFH.

To run the board scanner example type:

```
IC <CR>
```

The following figure is an implementation of the board scanner algorithm.

```
$title('ic - scan interconnect space and print map of system')
$compact

/****************************************************************************
 *              INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *          This software is supplied under the terms of a
 *          license agreement or nondisclosure agreement with
 *          Intel Corporation and may not be copied or disclosed
 *          except in accordance with the terms of that agreement.
 *
 *                    Copyright Intel Corporation 1987, 1988
 *                          All rights reserved
 *
 *  For Intel customers licensed for the iRMX II Operating
 *  System under an Intel Software License Agreement, this source code and
 *  object code derived therefrom are licensed for use on a single central
 *  processing unit for internal use only.  Necessary backup copies and
 *  multiple users are permitted.  Object Code derived from this source code
 *  is a Class I software product under the Intel Software License Agreement
 *  and is subject to the terms and conditions of that agreement.
 *
 *  For the right to make incorporations, or to transfer this software to
 *  third parties, contact Intel corporation.
 *
 */
/***************************************************************************/

/***********************************************************************
 *
 * MODULE NAME:  icscan
 *
 * DESCRIPTION: Scan the iPSB backplane.  Record each board instance and
 *              slot number in a system map.  Get local host slot number
 *              and id and record in system map.  Print system map on console.
 **********************************************************************/

ic: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(/rmx286/inc/error.lit)
$include(err.ext)
```

Figure 12-6. Implementation of a Board Scanner (continued)

```
DECLARE         /* literals */

    MAXSLOTS        LITERALLY  '20',    /* maximum number of slots */
    VENDORIDREG     LITERALLY  '00',    /* low byte of vendor id reg */
    BOARDIDREG      LITERALLY  '02H',   /* register offset of board id */
    BOARDIDLEN      LITERALLY  '10',    /* number of bytes in board id */
    BUFLEN          LITERALLY  '11',    /* length of board id rmx string */
    LOCALHOST       LITERALLY  '31',    /* selects local host interconnect */
    PSBCONTROLREC   LITERALLY  '06H',   /* psb control record value */
    PSBSLOTIDOFF    LITERALLY  '02H',   /* offset of slot id reg in psb control rec *
    NOEXCEPT        LITERALLY  '0',      /* no exception handling by system */
    RECNOTFOUND     LITERALLY  '0FFH',   /* indicates interconnect record not found *

    map_struc       LITERALLY 'STRUCTURE(
                                           id(BUFLEN)  BYTE,
                                          slot_num     BYTE)';
DECLARE
    con_tab(16) byte INITIAL('0','1','2','3','4','5','6','7','8',
                              '9','A','B','C','D','E','F');

$subtitle('find_record')
/*
 * PROC NAME:  find_record
 *
 * DESCRIPTION:
 *    Searches through the interconnect space of the specified board
 *     (slot_number) until either the record type passed is found or the
 *     EOT (end of template) record is found.  If the record is found,
 *     then its record type and offset are returned.  If the EOT record
 *     is found, then 0FFH is returned.
 *
 * CALL:  record_found = find_record(slot_number, record_type, rec_offset_ptr);
 * INPUTS: slot_number       BYTE containing slot number of board
 *         record_type       BYTE containing record type to find
 * RETURNS: record_found     byte indicating whether record was found
 *                           (contains 0FFH (EOT record type) if record not found)
 * OUTPUTS: record_offset_ptr  POINTER to WORD offset of record found
 *
 ******************************************************************************/
```

Figure 12-6. Implementation of a Board Scanner (continued)

```
   find_record: PROCEDURE(slot_number, record_type, rec_offset_ptr) BYTE PUBLIC
REENTRANT;

       DECLARE
           slot_number     BYTE,
           record_type     BYTE,
           rec_offset_ptr  POINTER,
           rec_offset      BASED rec_offset_ptr WORD,
           record_found    BYTE,
           status          WORD;
       DECLARE

           EOT_REC      LITERALLY '0FFH', /* end of template record */
           HDRRECLENGTH LITERALLY '20H',  /* number of registers in header rec */
           RECLENREG    LITERALLY '1';    /* record length register */

   /*
    *  Get record type for each record past the header
    *  record until the specified record is found
    */

   rec_offset = HDRRECLENGTH;
   record_found = rq$get$interconnect(slot_number, rec_offset, @status);
   CALL error$check(100, status);
   DO WHILE (record_found <> record_type) AND (record_found <> EOT_REC);
       /*
        * Get offset for next record by adding length of current record
        *  to previous record offset (add 2 to account for type and length
register)
        */
       rec_offset = rec_offset + 2 + rq$get$interconnect(slot_number,
                                         rec_offset + RECLENREG,
                                         @status);
       CALL error$check(110, status);
       record_found = rq$get$interconnect(slot_number, rec_offset, @status);
       CALL error$check(120, status);
       END;

   RETURN(record_found);

   END find_record;
```

Figure 12-6. Implementation of a Board Scanner (continued)

```
$subtitle('out_byte - print a byte on the console')
/************************************************************************
 *
 * PROC NAME: out_byte
 *
 * DESCRIPTION: Write the hex representation of hex_byte to the console.
 *
 * CALL: CALL out_byte(hex_byte)
 *
 * INPUT: hex_byte - byte whose hex value is to be printed
 *
 * GLOBALS:
 *
 ************************************************************************/

out_byte: procedure(hex_byte) public;

DECLARE
   hex_byte     BYTE,
   hex_buf(3)   BYTE,
   cur_byte     BYTE,
   status       WORD;

   cur_byte = shr(hex_byte, 4);
   hex_buf(0) = 2;
   hex_buf(1) = con_tab(cur_byte);
   cur_byte = hex_byte AND 0FH;
   hex_buf(2) = con_tab(cur_byte);
   call rqc$send$eo$response(NIL,0,@hex_buf,@status);
END out_byte;

$subtitle('print_map - print system map')
/************************************************************************
 *
 * PROC NAME: print_map
 *
 * ABSTRACT: Write system map to console
 *
 * CALL: CALL printmap(sys_map_ptr);
 *
 * INPUT: sys_map_ptr - pointer to system map
 *
 * GLOBALS:
 *
 * CALLS:  rqc$send$eo$response, out_byte
 *
 ************************************************************************/
```

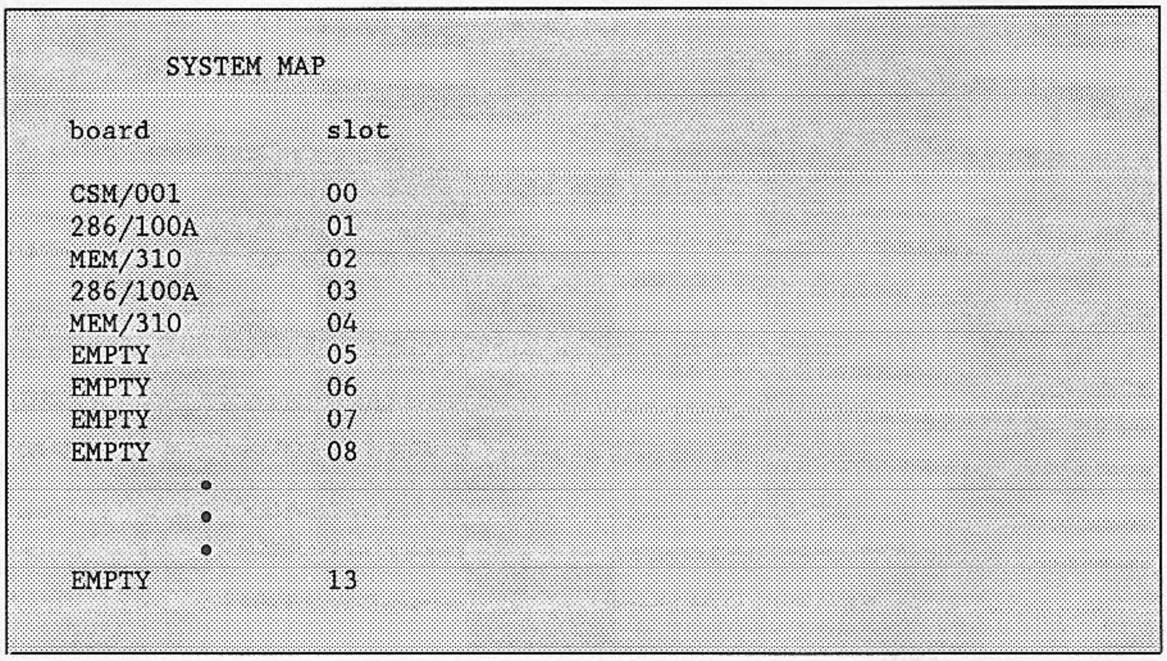
Figure 12-6. Implementation of a Board Scanner (continued)

```
print_map: PROCEDURE(sys_map_ptr) PUBLIC;

   DECLARE       /* params */

       sys_map_ptr   POINTER;

   DECLARE       /* locals */

       sys_map  BASED  sys_map_ptr(MAXSLOTS) map_struc,  /* system map */
       i                                     BYTE,       /* local index */
       status                                WORD;

   CALL rqc$send$eo$response(NIL,0,@(2,0dh,0ah),@status);
   CALL rqc$send$eo$response(NIL,0,@(17,'     SYSTEM MAP',0dh,0ah),@status);
   CALL rqc$send$eo$response(NIL,0,@(23,'board           slot',0dh,0ah,0dh,0ah),
                                    @status);
   DO i = 0 to MAXSLOTS -1;
       CALL rqc$send$eo$response(NIL,0,@sys_map(i).id,@status);
       CALL rqc$send$eo$response(NIL,0,@(5,'      '),@status);
       CALL out_byte(sys_map(i).slot_num);
       CALL rqc$send$eo$response(NIL,0,@(2,0DH,0Ah),@status);
   END;

END print_map;

DECLARE          /* globals */

   iPSB_slot             BYTE,     /* psb slot currently scanned */
   status                WORD,
   count                 BYTE,     /* index into id string */
   vendor_id_lo          BYTE,     /* low byte of vendor id */
   vendor_id_hi          BYTE,
   id_char               BYTE,     /* character in board id */
   local_slot            BYTE,     /* slot number of local host */
   psb_reg_off           WORD,     /* offset of psb control record */
   r_type                BYTE,     /* record type returned by find_record */
   sys_map(MAXSLOTS)     map_struc;/* map of boards in system */

             /* begin main */

   CALL set$exception(NOEXCEPT);
   DO iPSB_slot = 0 TO MAXSLOTS - 1;
       sys_map(iPSB_slot).slot_num = iPSB_slot;
       vendor_id_lo = rq$get$interconnect(iPSB_slot, VENDORIDREG, @status);
       CALL error$check(130, status);
```

Figure 12-6. Implementation of a Board Scanner (continued)

```
         *
         * Only check status after first call to get$interconnect,
         * to see if call is configured (no other error is returned
         * by get$interconnect)
         */
        vendor_id_hi = rq$get$interconnect(iPSB_slot, VENDORIDREG+1, @status);
        CALL error$check(140, status);
        /*
         * If vendor_id is not equal to 0, then there is a
         * board in this slot, so get the board's id
         */

        IF ((vendor_id_hi OR vendor_id_lo) <> 0) THEN DO;
            count = 0;
            sys_map(iPSB_slot).id(0) = BOARDIDLEN;
            DO WHILE (count < BOARDIDLEN);
                id_char = rq$get$interconnect(iPSB_slot, BOARDIDREG+count, @status)
                CALL error$check(150, status);
                IF (id_char <= '!') OR (id_char >= '}') THEN
                    id_char = ' ';
                sys_map(iPSB_slot).id(count+1) = id_char;
                count = count + 1;
            END;
        END;
        ELSE
            CALL movb(@(10,'EMPTY     '), @sys_map(iPSB_slot).id, BOARDIDLEN+1);
    END;
    /*
     *  Now get slot number and id of local host.  To access local intercon-
     *  nect, the special slot number, LOCALHOST, must be used
     */
    r_type = find_record(LOCALHOST,PSBCONTROLREC,@psb_reg_off);
    IF r_type <> RECNOTFOUND THEN DO;
        local_slot= rq$get$interconnect(LOCALHOST, psb_reg_off + PSBSLOTIDOFF,
                                                        @status);
        CALL error$check(160, status);
        local_slot = shr(local_slot,3);
        sys_map(local_slot).slot_num = local_slot;
        count = 0;
        DO WHILE (count < BOARDIDLEN);
            id_char = rq$get$interconnect(LOCALHOST, BOARDIDREG+count, @status);
            CALL error$check(170, status);
            IF (id_char <= '!') OR (id_char >= '}') THEN
                id_char = ' ';
            sys_map(local_slot).id(count+1) = id_char;
            count = count + 1;
        END;
    END;
    CALL print_map(@sys_map);
    CALL rq$exit$io$job(0,NIL,@status);
END ic;
```

Figure 12-6. Implementation of a Board Scanner



```
          SYSTEM MAP

     board               slot

     CSM/001             00
     286/100A            01
     MEM/310             02
     286/100A            03
     MEM/310             04
     EMPTY               05
     EMPTY               06
     EMPTY               07
     EMPTY               08
              •
              •
              •
     EMPTY               13
```

Figure 12-7. Sample Screen Output From the Interconnect Example

Figure 12-7 shows a sample output of the board scanner example.

12.5.6.2 Creating a Port for Message Sending and Receiving

Once you have information on what boards are in your system, the next step is to create a port for message passing and associate a buffer pool with it. The following example creates a buffer pool, releases a number of 1K buffers to it, and then creates a data transport type port and returns a TOKEN to use as a reference to the port.

This module is not run from the Human Interface, it is called by the other modules described in this chapter.

```
$title('crport - create a port and attach a buffer pool to it')
$compact

/****************************************************************************
 *              INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *           This software is supplied under the terms of a
 *           license agreement or nondisclosure agreement with
 *           Intel Corporation and may not be copied or disclosed
 *           except in accordance with the terms of that agreement.
 *
 *                 Copyright Intel Corporation 1987, 1988
 *                           All rights reserved
 *
 *    For Intel customers licensed for the iRMX II Operating
 *    System under an Intel Software License Agreement, this source code and
 *    object code derived therefrom are licensed for use on a single central
 *    processing unit for internal use only.  Necessary backup copies and
 *    multiple users are permitted.  Object Code derived from this source code
 *    is a Class I software product under the Intel Software License Agreement
 *    and is subject to the terms and conditions of that agreement.
 *
 *    For the right to make incorporations, or to transfer this software to
 *    third parties, contact Intel corporation.
 *
 */
/***************************************************************************/

crport: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(/rmx286/inc/error.lit)
$include(dcom.lit)
$include(err.ext)

DECLARE

        NOEXCEPT    LITERALLY    '0';    /* no exception handling by system */
```

Figure 12-8. Creating a Data Transport Protocol Port (continued)

```
$eject
$subtitle('create$buf$pool')
/************************************************************************
 *
 *     PROC NAME: create$buf$pool
 *
 * DESCRIPTION:  Create a buffer pool with the attributes passed by the
 *               caller. Create an initial number of 1K buffers and
 *               release them to the buffer pool.  Return a token for
 *               the buffer pool to the caller.
 *
 * CALL: buf$pool$tok = create$buf$pool(max_bufs, init_num_bufs,
 *                                      attrs, status_ptr);
 * INPUTS:  max_bufs - maximum number of buffers for buffer pool
 *          init_num_bufs - initial number of buffers for buffer
 *          pool.
 *          attrs - buffer pool creation attributes
 *          status_ptr - points to a status word
 *
 * RETURNS:  buf$pool$tok - token for newly created buffer pool
 *
 ************************************************************************/
create$buf$pool: PROCEDURE(max_bufs, init_num_bufs, attrs, status_ptr) TOKEN
PUBLIC;

   DECLARE         /* Parameters */

       max_bufs        WORD,       /* maximum number of buffers in buffer pool */
       init_num_bufs   WORD,       /* initial number of buffers in pool */
       attrs           WORD,       /* buffer pool creation attributes */
       status_ptr      POINTER;    /* exception pointer */

   DECLARE         /* Local Variables */

       status    BASED status_ptr WORD ,
       buf_pool                   TOKEN, /* buffer pool complete with buffers */
       buf_tok                    TOKEN, /* buffer token */
       i                          WORD;  /* local index */
   DECLARE         /* Literals */

       BUFSIZE     LITERALLY  '1024',     /* buffer size */
       BFLAGS      LITERALLY  '010B';     /* single buffer, don't release */

   buf_pool = rq$create$buffer$pool(max_bufs, attrs, status_ptr);
   CALL error$check(10, status);
```

Figure 12-8. Creating a Data Transport Protocol Port (continued)

```
    DO i = 1 to init_num_bufs;
        buf_tok = rq$create$segment(BUFSIZE, status_ptr);
        CALL error$check(20, status);
        CALL rq$release$buffer(buf_pool, buf_tok, BFLAGS, status_ptr);
        CALL error$check(30, status);
    END;
    RETURN buf_pool;

END create$buf$pool;

$eject
$subtitle('get$dport')
/*********************************************************************
 *
 *      PROC NAME: get$dport
 *
 *      DESCRIPTION: This procedure creates a port for data transport service.
 *                   A buffer pool is created and attached to the port.
 *                   A token for the newly created port and buffer pool are
 *                   returned to the caller.  If either port or buffer
 *                   creation fails, the call returns with neither a buffer
 *                   pool or port created.
 *
 *      CALL: dport$tok = get$dport(port_num,buf_pool_ptr, b_attrs,
 *                          status_ptr)
 *
 *      INPUTS:     port_num - port number assigned to newly created port
 *                  b_attrs - buffer pool creation attributes
 *                  status_ptr - points to status word
 *      OUTPUTS:    buf_pool_ptr - points to buffer pool token attached to
 *                          the newly created port
 *      RETURNS:    dport$tok - token to newly created port
 *
 *********************************************************************/

get$dport: PROCEDURE(port_num, buf_pool_ptr,b_attrs, status_ptr) TOKEN PUBLIC;

   DECLARE     /* Parameters */

        port_num        WORD,               /* port id for new port */
        buf_pool_ptr    POINTER,            /* points to buffer pool */
        b_attrs         WORD,               /* buffer pool creation attributes */
        status_ptr      POINTER;
```

Figure 12-8. Creating a Data Transport Protocol Port (continued)

```
    DECLARE     /* Literals */

        NUM_TRANS   LITERALLY   '10',    /* max number outstanding trans at port */
        DATACOM     LITERALLY   '2',     /* indicates data com port */
        PFLAGS      LITERALLY   '0',     /* fifo, fragmentation flags */
        MAXBUFS     LITERALLY   '30',    /* maximum # buffers in pool */
        INITBUFS    LITERALLY   '10';    /* initial number of buffers */

    DECLARE     /* locals */

        bpool BASED buf_pool_ptr TOKEN,
        port_t                   TOKEN,        /* local port */
        bufpool_t                TOKEN,        /* buffer pool with initial  alloc of
                                                  buffers */
        port_info                port_info_s,
        loc_status               WORD,         /* local status word */
        status  BASED status_ptr WORD;

                /* Begin get$dport */

        port_info.port_id= port_num;
        port_info.type = DATACOM;
        port_info.reserved = 0;
        port_info.flags = PFLAGS;
        port_t = rq$create$port(NUM_TRANS, @port_info, status_ptr);
        CALL error$check(40, status);
        bufpool_t = create$buf$pool(MAXBUFS, INITBUFS, b_attrs, status_ptr);
        CALL error$check(50, status);
        bpool = bufpool_t;
        CALL rq$attach$buffer$pool(bufpool_t, port_t, status_ptr);
        CALL error$check(60, status);
        RETURN port_t;

END get$dport;

END crport;
```

Figure 12-8. Creating a Data Transport Protocol Port

12.5.6.3 Sending Data Using RQ$SEND$RSVP

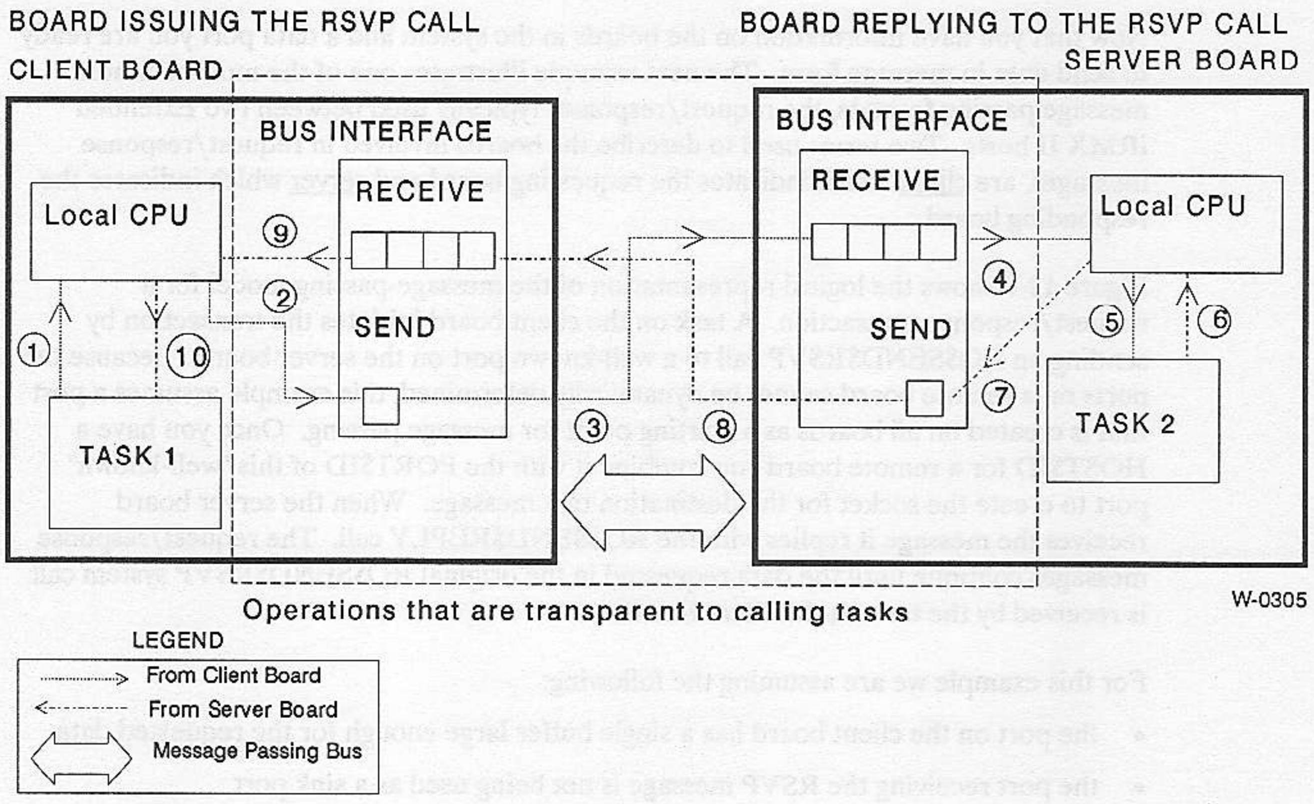
Now that you have information on the boards in the system and a data port you are ready to send data in message form. The next example illustrates one of the most common message passing formats, the request/response, typically used between two Extended iRMX II hosts. Two terms used to describe the boards involved in request/response messages, are client which indicates the requesting board and server which indicates the responding board.

Figure 12-9 shows the logical representation of the message-passing model for a request/response transaction. A task on the client board initiates the transaction by sending an RQ$SEND$RSVP call to a well-known port on the server board. Because the ports on a remote board cannot be dynamically determined, this example assumes a port that is created on all boards as a starting point for message passing. Once you have a HOST$ID for a remote board you combine it with the PORT$ID of this "well-known" port to create the socket for the destination of a message. When the server board receives the message it replies with the RQ$SEND$REPLY call. The request/response messages continue until the data requested in the original RQ$SEND$RSVP system call is received by the task on the client board.

For this example we are assuming the following:

- the port on the client board has a single buffer large enough for the requested data
- the port receiving the RSVP message is not being used as a sink port

Figure 12-10 is an algorithm for this transaction and Figure 12-4 shows the physical location of the boards in a system.



1. The task on the Client board issues an RQ$SEND$RSVP call. In an RSVP/REPLY transaction, the board that issues the send RSVP is the client; the board that replies is the server.
2. The Nucleus Communication Service turns the information in the "RSVP" system call into a message, and sets up the buffer space for the expected reply.
3. The MPC sends the message across a message passing bus to the remote agent specified in the RSVP system call.
4. The CPU on the server board receives a PIC interrupt informing it that a MULTIBUS II message has been received.
5. The Nucleus Communication Service on the server board directs the message to the appropriate port (and therefore task.)
6. Task 2 responds with an RQ$SEND$REPLY system call that contains information about the data being sent.
7. The Nucleus Communication Service on the server board turns the information in the RQ$SEND$REPLY call into a message that is sent by the MPC.
8. The message travels across the message passing bus, an operation that is transparent to the operating systems on both boards.
9. The MPC on the client board places the message into the buffer that was set up in step two, and then sends an interrupt to the CPU, informing it of the completion of the message transaction.
10. The Nucleus Communication Service on the client board directs the message to the correct task using the PORT$ID. The CPU on the client board is "aware" of only the operations performed in steps 1, 2, 9, and 10.

Figure 12-9. An RSVP/REPLY Transaction between Two Extended iRMX® II Hosts

```
Client board
    Call an external procedure called get$dport that returns a TOKEN
      for the local port to be used in the RQ$SEND$RSVP call.
    Initialize the socket structure, declared externally.
    Set the message size to be zero length.
    Equate the global variable rsvp_size to the LITERAL RSVPB (128 bytes.)
    Issue the RSVP system call using the previously initialized variables.
    Use the RQ$RECEIVE$REPLY system call to wait for an answer.
    Send the reply message, "This is a send$reply" message" to the console screen.
    Exit from the example.
```

Figure 12-10. Algorithm for RQ$SEND$RSVP Example

```
Server board
    Call an external procedure, get$dport, that returns a TOKEN to
        be used in the RQ$RECEIVE and RQ$SEND$REPLY calls.
    Perform an RQ$RECEIVE using the TOKEN returned from get$dport
    Perform an RQ$SEND$REPLY on successful completion of the RQ$RECEIVE
        IF the data arrives correctly, msg_ptr <> NIL
            Return the buffer to the buffer pool
    End server procedure
```

Figure 12-11 Algorithm for Server Board

This example must be run with the following example shown in Figure 12-13. To run these two examples, first on the host in slot five type:

```
RCVMSG <CR>
```

Then on the host in slot one type:

```
SNDRSVP <CR>
```

```
$title('sndrsvp - initiate a request-response transaction')
$compact

/****************************************************************************
 *              INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *         This software is supplied under the terms of a
 *         license agreement or nondisclosure agreement with
 *         Intel Corporation and may not be copied or disclosed
 *         except in accordance with the terms of that agreement.
 *
 *                    Copyright Intel Corporation 1987, 1988
 *                              All rights reserved
 *
 *   For Intel customers licensed for the iRMX II Operating
 *   System under an Intel Software License Agreement, this source code and
 *   object code derived therefrom are licensed for use on a single central
 *   processing unit for internal use only.  Necessary backup copies and
 *   multiple users are permitted.  Object Code derived from this source code
 *   is a Class I software product under the Intel Software License Agreement
 *   and is subject to the terms and conditions of that agreement.
 *
 *   For the right to make incorporations, or to transfer this software to
 *   third parties, contact Intel corporation.
 */
/***************************************************************************/


/***********************************************************************
 *
 *  MODULE NAME: sndrsvp
 *
 *  DESCRIPTION: Send a transaction request to a well-known socket. The request
 *               is sent as an unsolicited message, ie with no data part.  Wait
 *               for a response and print the message on the console.
 *
 ***********************************************************************/

sndrsvp: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(dcom.ext)
$include (dcom.lit)
$include(/rmx286/inc/error.lit)
$include(err.ext)
```

Figure 12-12. Client Board Code for RQ$SEND$REPLY Example (Continued)

```
DECLARE        /* Literals */

    REMPORT    LITERALLY  '801H',      /* Port id of remote port */
    REMHOSTID  LITERALLY  '5',         /* hostid of remote host */
    CONBUF     LITERALLY  '20',        /* control buffer size */
    RSVPB      LITERALLY  '128',       /* rsvp buffer size */
    TSTPORT    LITERALLY  '801H',      /* well-known port */
    NOEXCEPT   LITERALLY  '0',         /* no exception handling by system */
    SFLAGS     LITERALLY  '00000B';    /* data buffer, synch, rcvreply flags*/

    DECLARE    /* Global vars */

        status              WORD,
        port_t              TOKEN,                  /* Token for local port */
        messock             socket,                 /* socket to which message is sent *
        msock               DWORD AT (@messock),    /* dword alias for messock */
        con_buf (CONBUF)    BYTE,                   /* control buffer */
        rsvp_buf (RSVPB)    BYTE,                   /* rsvp buffer */
        mess_size           DWORD,                  /* number of bytes in data message *
        rsvp_size           DWORD,                  /* rsvp buffer size */
        rsvp_ptr            POINTER,                /* points to rsvp message */
        info                rec_info,               /* receive info block */
        buf_pool            TOKEN,                  /* buffer pool attached to port */
        trans_id            WORD;                   /* transaction id */

    CALL set$exception(NOEXCEPT);
    port_t = get$dport(TSTPORT, @buf_pool, CHAIN, @status);
    messock.host_id = REMHOSTID;
    messock.port_id = REMPORT;
    mess_size = 0;
    rsvp_size = RSVPB;
    trans_id = rq$send$rsvp(port_t,msock, @con_buf, NIL,
                        mess_size, @rsvp_buf, rsvp_size, SFLAGS, @status);
    CALL error$check(100, status);
    rsvp_ptr = rq$receive$reply(port_t, trans_id, WAITFOREVER, @info, @status);
    CALL error$check(110, status);
    call rqc$send$eo$response(NIL,0,@rsvp_buf,@status);
    CALL error$check(120, status);
    call rq$exit$io$job(0,NIL,@status);

end sndrsvp;
```

Figure 12-12. Client Board Code for RQ$SEND$RSVP Example

```
$title('rcvrsvp - respond to a request-response transaction')
$compact

/**************************************************************************
 *          INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *          This software is supplied under the terms of a
 *          license agreement or nondisclosure agreement with
 *          Intel Corporation and may not be copied or disclosed
 *          except in accordance with the terms of that agreement.
 *
 *                      Copyright Intel Corporation 1987, 1988
 *                              All rights reserved
 *
 *   For Intel customers licensed for the iRMX II Operating
 *   System under an Intel Software License Agreement, this source code and
 *   object code derived therefrom are licensed for use on a single central
 *   processing unit for internal use only.  Necessary backup copies and
 *   multiple users are permitted.  Object Code derived from this source code
 *   is a Class I software product under the Intel Software License Agreement
 *   and is subject to the terms and conditions of that agreement.
 *
 *   For the right to make incorporations, or to transfer this software to
 *   third parties, contact Intel corporation.
 *
 */
/*************************************************************************/

/*********************************************************************
 *
 *     MODULE NAME: rcvrsvp
 *
 *     DESCRIPTION: When a message is received, send a response via rq$send$reply
 *                  and exit.
 *
 *
 *********************************************************************/

rcvrsvp: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(dcom.ext)
$include(dcom.lit)
$include(/rmx286/inc/error.lit)
$include(err.ext)
```

Figure 12-13. The Server Board Code to Receive and Answer an RSVP Message (Continued)

```
    DECLARE                 /* Literals */

        TSTPORT     LITERALLY   '801H',     /* well-known port */
        SFLAGS      LITERALLY   '00000B',   /* data buffer, synchronous flags*/
        NOEXCEPT    LITERALLY   '0';        /* no exception handling by system */


    DECLARE                 /* Global vars */

        status          WORD,
        port_t          TOKEN,      /* Token for local port */
        info            rec_info,   /* info block on message received */
        buf_pool        TOKEN,      /* buffer pool attached to port */
        mes_buf(*)      BYTE initial (30,'This is a send$reply message',0dh,0ah),
        tran_id         WORD,
        con_buf (20) BYTE,          /* control message buffer */
        msg_ptr         POINTER;    /* pointer to received message */


    CALL set$exception(NOEXCEPT);
    port_t = get$dport(TSTPORT, @buf_pool, CHAIN, @status);
    msg_ptr = rq$receive(port_t, WAITFOREVER, @info, @status);
    CALL error$check(100, status);
    tran_id = rq$send$reply(port_t, info.rem$socket,
                            info.trans$id, @con_buf,
                            @mes_buf, SIZE(mes_buf), SFLAGS, @status);
    CALL error$check(110, status);
    IF msg_ptr <> NIL THEN DO;
        CALL rq$release$buffer(buf_pool, selector$of(msg_ptr), (info.flags AND 3),
                               @status);
        CALL error$check(100, status);
    END;
    CALL rq$exit$io$job(0,NIL,@status);
END rcvrsvp;
```

Figure 12-13. The Server Board Code to Receive and Answer an RSVP Message

12.5.6.4 Sending a Data Chain Message

This section presents an example of sending and receiving a message that is in data chain form. The example is presented in two modules, one that sends the data chain and one that receives it. Note that a port's ability to receive messages in data chain form is set according to the attributes of the port's associated **buffer pool**.

This example must be run with the following example shown in Figure 12-15. To run these example two commands must be typed, one on each host terminal. First, on the host in slot five, type:

```
RCVMSG <CR>
```

Second, on the host in slot one, type:

```
DCSNDMSG <CR>
```

The host terminal in slot five will display:

```
This is a data chain message sent by server.
```

```
$title('dcsndmsg - send a data chain message to a known port')
$compact

/*****************************************************************************
 *                INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *            This software is supplied under the terms of a
 *            license agreement or nondisclosure agreement with
 *            Intel Corporation and may not be copied or disclosed
 *            except in accordance with the terms of that agreement.
 *
 *                       Copyright Intel Corporation 1987, 1988
 *                              All rights reserved
 *
 *    For Intel customers licensed for the iRMX II Operating
 *    System under an Intel Software License Agreement, this source code and
 *    object code derived therefrom are licensed for use on a single central
 *    processing unit for internal use only.  Necessary backup copies and
 *    multiple users are permitted.  Object Code derived from this source code
 *    is a Class I software product under the Intel Software License Agreement
 *    and is subject to the terms and conditions of that agreement.
 *
 *    For the right to make incorporations, or to transfer this software to
 *    third parties, contact Intel corporation.
 *
 *
*****************************************************************************/
```

Figure 12-14. Data Chain Send (continued)

```
/*********************************************************************
 *
 *     MODULE NAME: dcsndmsg
 *
 *     DESCRIPTION: Send a data chain message.
 *
 *********************************************************************/

dcsndmsg: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(dcom.ext)
$include (dcom.lit)
$include(/rmx286/inc/error.lit)
$include(err.ext)

DECLARE         /* Literals */

      dc_el     LITERALLY 'STRUCTURE( /* data chain element */
                                      b_size      WORD,     /* buffer size */
                                      buf_ptr     POINTER, /* buffer pointer */
                                      res         WORD)',   /* reserved */

   REMPORT      LITERALLY '801H',    /* Port id of remote port */
   REMHOST      LITERALLY '05',      /* Host id of remote host */
   CONBUF       LITERALLY  '16',     /* size of a control buffer */
   TSTPORT      LITERALLY  '801H',   /* well-known port */
   MSIZE        LITERALLY  '46',     /* message size */
   BUFSIZE      LITERALLY  '100',    /* buffer size */
   NOEXCEPT     LITERALLY  '0',      /* no exception handling by system */
   DCBUFSIZE    LITERALLY  '8';      /* data chain buffer size */

   DECLARE      /* Global vars */

       status      WORD,
       port_t      TOKEN,                 /* Token for local port */
       messock     socket,                /* socket to which message is sent */
       msock       DWORD AT (@messock),   /* dword alias for messock */
       con_buf (CONBUF) BYTE,             /* control buffer */
       mess_size   WORD,                  /* number of bytes in data message */
       bpool       TOKEN,                 /* buffer pool attached to port */
       offset      WORD,                /* buffer offset where chain buffer starts *
       sflags      WORD,                  /* transmission flags */
       dc_seg_size WORD,                  /* segment size for data chain */
       dc_seg_t    TOKEN,                 /* token for data chain segment */
       dc_ptr      POINTER,               /* pointer to data chain segment */
       d_chain based dc_ptr(1) dc_el,     /* data chain */
       dc_idx      WORD,                  /* data chain index */
       trans_id    WORD;                  /* transaction id */
```

Figure 12-14. Data Chain Send (continued)

```
DECLARE

    dc_buf (BUFSIZE) BYTE INITIAL
     (45,'This is a data chain message sent by server',0dh,0ah);

CALL set$exception(NOEXCEPT);
port_t = get$dport(TSTPORT, @bpool, CHAIN, @status);
messock.host_id = REMHOST;
messock.port_id = REMPORT;

/* create data chain with at least enough blocks for each message
   buffer + a terminating block */

mess_size = SIZE(dc_buf);
/*
 * Calculate the size of the segment that will contain the data chain.
 * The message is divided into pieces whose size is DCBUFSIZE so the total
 * number of elements in the data chain is mess_size/DCBUFSIZE + 2.
 * The additional 2 includes one possible piece of the message less than
 * DCBUFSIZE and the terminating data chain element.
 */

dc_seg_size = (mess_size/DCBUFSIZE + 2)*(size(d_chain));
dc_seg_t = rq$create$segment(dc_seg_size, @status);
dc_ptr = build$ptr(dc_seg_t,0);

/* Fill in the fields of the data blocks for each buffer containing
   a part of the message */

offset = 0;
dc_idx = 0;
DO WHILE offset < mess_size;
    d_chain(dc_idx).b_size = DCBUFSIZE;
    d_chain(dc_idx).buf_ptr = @dc_buf(offset);
    offset = offset + DCBUFSIZE;
    dc_idx = dc_idx + 1;
END;
d_chain(dc_idx).b_size = 0;
d_chain(dc_idx).buf_ptr = NIL;
/* send data chain */

sflags = DATACHAIN OR SYNCHTRANS;
trans_id = rq$send(port_t,msock, @con_buf, @d_chain,
                        mess_size, sflags, @status);
CALL error$check(100, status);
CALL rq$exit$io$job(0,NIL,@status);

END dcsndmsg;
```

Figure 12-14. Data Chain Send

```
$title('dcrcvmsg - receive a 2K data chain message')
$compact

/***************************************************************************
 *                INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *       This software is supplied under the terms of a
 *       license agreement or nondisclosure agreement with
 *       Intel Corporation and may not be copied or disclosed
 *       except in accordance with the terms of that agreement.
 *
 *                Copyright Intel Corporation 1987, 1988
 *                          All rights reserved
 *
 *   For Intel customers licensed for the iRMX II Operating
 *   System under an Intel Software License Agreement, this source code and
 *   object code derived therefrom are licensed for use on a single central
 *   processing unit for internal use only.  Necessary backup copies and
 *   multiple users are permitted.  Object Code derived from this source code
 *   is a Class I software product under the Intel Software License Agreement
 *   and is subject to the terms and conditions of that agreement.
 *
 *   For the right to make incorporations, or to transfer this software to
 *   third parties, contact Intel corporation.
 */
/*************************************************************************/

/**********************************************************************
 *
 *     MODULE NAME: dcrcvmsg
 *
 *     DESCRIPTION: When a message is received, determine whether it is in data
 *                  chain or buffer form.  If data chain, compress the chain into
 *                  a single segment.  Expect a 2K message with a printable
 *                  part at the first and second 1K + 2 boundaries.  Write the
 *                  printable part to the console.
 *
 **********************************************************************/

dcrcvmsg: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(dcom.ext)
$include(dcom.lit)
$include(/rmx286/inc/error.lit)
$include(err.ext)
```

Figure 12-15. Receive a Message in Data Chain Form (continued)

```
DECLARE                /* Literals */

    TSTPORT      LITERALLY   '801H',          /* well-known port */
   NOEXCEPT     LITERALLY    '0';           /* no exception handling by system */

DECLARE     /* Global vars */

        status       WORD,
        port_t       TOKEN,                 /* Token for local port */
        local_host   WORD,                  /* local host id */
        info         rec_info,              /* info block on message received */
        bpool        TOKEN,                 /* buffer pool */
        dcmsg_ptr    POINTER,               /* pointer to data chain message */
        msg_ptr      POINTER,               /* pointer to received message */
        msg BASED dcmsg_ptr (1) BYTE;

$subtitle('get$dc$data')
/**********************************************************************
 *
 *     PROC NAME: get$dc$data
 *
 *     DESCRIPTION: This procedure takes a data chain and copies the data described
 *                  by it into a single segment.  This procedure only works if the
 *                  data is less than 64K in size.  Data chains can describe data
 *                  greater than 64K.
 *
 *     CALL: mbuf_ptr = get$dc$data(dc_ptr, status_ptr)
 *
 *     INPUTS:  dc_ptr - points to a data chain
 *              status_ptr - points to a status word
 *
 *     RETURNS: mbuf_ptr - points to a segment containing the data
 *                         described by a data chain
 **********************************************************************/
get$dc$data: PROCEDURE(dc_ptr, status_ptr) POINTER PUBLIC;

   DECLARE          /* Params */

       dc_ptr       POINTER, /* points to data chain */
       status_ptr   POINTER; /* points to status word */

   DECLARE          /* Locals */
```

Figure 12-15. Receive a Message in Data Chain Form (continued)

```
        dc BASED dc_ptr (1)       blk_struc,
        status BASED status_ptr WORD,
        num_bytes                 WORD,    /* number of bytes in data chain */
        cpybuf_tok                TOKEN,   /* buffer to hold data chain data */
        cpybuf_ptr                POINTER, /* points to cpybuf */
        cpybuf BASED cpybuf_ptr c_buf,
        i                         WORD,    /* local index */
        cpyidx                    WORD;    /* index into cpybuf */

    num_bytes = 0;
    i = 0;

    /* get the size of the data described by the data chain */

    DO WHILE dc(i).b_size <> 0;

        num_bytes = num_bytes + dc(i).b_size;
        i=i+1;
    END;
    /* add 2 to num_bytes for the size field in c_buf */
    num_bytes = num_bytes + 2;
    cpybuf_tok = rq$create$segment(num_bytes, status_ptr);
    CALL error$check(100, status);
    cpybuf_ptr = build$ptr(cpybuf_tok, 0);
    cpybuf.size = num_bytes - 2;

    i = 0;
    cpyidx = 0;
    DO WHILE dc(i).b_size <> 0;
        CALL movb(dc(i).buf_ptr, @cpybuf.buf(cpyidx), dc(i).b_size);
        cpyidx = cpyidx + dc(i).b_size;
        i = i + 1;
    END;

    RETURN cpybuf_ptr;
END get$dc$data;


            /* Start main */

    CALL set$exception(NOEXCEPT);
    port_t = get$dport(TSTPORT, @bpool, CHAIN, @status);
    msg_ptr = rq$receive(port_t, WAITFOREVER, @info, @status);
    CALL error$check(110, status);
    IF (info.flags AND DATACHAIN) = DATACHAIN THEN DO;
        dcmsg_ptr = get$dc$data(msg_ptr, @status);
        CALL error$check(120, status);
```

Figure 12-15. Receive a Message in Data Chain Form (continued)

```
        /*
         * print message that was contained at start of the buffer described
         * by the first element in the data chain
        */
        CALL rqc$send$eo$response(NIL,0,@msg(2),@status);
        CALL error$check(130, status);
        /*
         * print message that was contained at start of the buffer described
         * by the second element in the data chain
         *
         */
        CALL rqc$send$eo$response(NIL,0,@msg(1026),@status);
        CALL error$check(140, status);
    END;
    ELSE DO;
        CALL rqc$send$eo$response(NIL,0,msg_ptr,@status);
        CALL error$check(150, status);
    END;
    CALL rq$release$buffer(bpool, SELECTOR$OF(msg_ptr), (info.flags AND 3),
                           @status);
    CALL rq$exit$io$job(0,NIL,@status);
  END dcrcvmsg;
```

Figure 12-15. Receive a Message in Data Chain Form

12.5.6.5 Sending a Message in Fragments

This section presents an example of sending and receiving a message that is broken into fragments. The example is presented in two modules, one that sends the fragmented message and one that receives it. Note that a port's ability to receive messages in fragment form is set according to the attributes given to the port at the time of its creation.

This example must be run with the RSVP procedure shown in Figure 12-12. To run this example two commands must be typed, one on each host terminal. First, on the host in slot five, type:

```
SNDFRAG <CR>
```

This procedure will break the data into fragments and send them to the processor board in slot one.

Second, on the host in slot one, type:

```
SNDRSVP <CR>
```

This procedure will receive the fragmented data and display in on the terminal.

The host terminal in slot one will display, "`This is a reply sent in fragments.`"

```
$title('sndfrag - send a fragmented message')
$compact

/******************************************************************************
 *              INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *          This software is supplied under the terms of a
 *          license agreement or nondisclosure agreement with
 *          Intel Corporation and may not be copied or disclosed
 *          except in accordance with the terms of that agreement.
 *
 *                    Copyright Intel Corporation 1987, 1988
 *                          All rights reserved
 *
 *   For Intel customers licensed for the iRMX II Operating
 *   System under an Intel Software License Agreement, this source code and
 *   object code derived therefrom are licensed for use on a single central
 *   processing unit for internal use only.  Necessary backup copies and
 *   multiple users are permitted.  Object Code derived from this source code
 *   is a Class I software product under the Intel Software License Agreement
 *   and is subject to the terms and conditions of that agreement.
 *
 *   For the right to make incorporations, or to transfer this software to
 *   third parties, contact Intel corporation.
 *
 */
/*****************************************************************************/

/*************************************************************************
 *
 *     MODULE NAME: sndfrag
 *
 *     DESCRIPTION: Receive a transaction request, send the reply as a fragmented
 *                  message.
 *
 *************************************************************************/

sndfrag: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(dcom.ext)
$include(dcom.lit)
$include(/rmx286/inc/error.lit)
$include(err.ext)
```

Figure 12-16. Send a Message in Fragments (continued)

```
DECLARE                  /* Literals */

     FRAGLEN     LITERALLY   '8',        /* fragmentation buffer length */
     TSTPORT     LITERALLY   '801H',     /* well-known port */
     EOTFLAGS    LITERALLY   '00000B',   /* send$reply flags for buffer, synch
                                            tran and eot */
     NOEXCEPT    LITERALLY   '0',        /* no exception handling by system */
     NOTEOTFLAGS LITERALLY   '0200H';    /* same as above except not eot */

DECLARE                  /* Global vars */

     status                WORD,
     port_t                TOKEN,     /* Token for local port */
     info                  rec_info,  /* info block on message received */
     buf_pool              TOKEN,     /* buffer pool attached to port */
     mes_buf(*)            BYTE initial (35,'This is a reply sent in
                                          fragments',0dh,0ah),
     mes_idx               WORD,      /* mes_buf index */
     mes_size              WORD,      /* size of mes_buf */
     frag_size             WORD,      /* size of fragment sent */
     sflags                WORD,      /* send message flags */
     tran_id               WORD,      /* transaction id */
     con_buf (20)          BYTE,      /* control message buffer */
     msg_ptr               POINTER;   /* pointer to received message */

CALL set$exception(NOEXCEPT);
port_t = get$dport(TSTPORT, @buf_pool, NOCHAIN, @status);
msg_ptr = rq$receive(port_t, WAITFOREVER, @info, @status);
CALL error$check(100, status);

IF info.status = E$OK THEN DO;
     mes_size = size(mes_buf);
     mes_idx = 0;
     sflags = NOTEOTFLAGS;
     frag_size = FRAGLEN;
```

Figure 12-16. Send a Message in Fragments (continued)

```
            /* Break message into fragments and send them */
            DO WHILE mes_idx < mes_size;
                IF mes_idx + FRAGLEN > mes_size THEN DO;
                    frag_size = mes_size - mes_idx;
                    sflags = EOTFLAGS;
                END;
                tran_id = rq$send$reply(port_t, info.rem$socket, info.trans$id,
                                        @con_buf, @mes_buf(mes_idx), frag_size,
                                        sflags, @status);
                CALL error$check(110, status);
                mes_idx = mes_idx + FRAGLEN;
            END;
            IF msg_ptr <> NIL THEN DO;
                CALL rq$release$buffer(buf_pool, selector$of(msg_ptr), 0, @status);
                CALL error$check(110, status);
            END;
        END;
        CALL rq$exit$io$job(0,NIL,@status);
    END sndfrag;
```

Figure 12-16. Send a Message in Fragments

12.5.6.6 Receiving a Message in Fragment Form

This section presents an example of sending a message and receiving it in fragment form. The example is presented in two modules, one, SFRAG, that initiates a transaction which forces receiving to be done in fragment form. The other module, RCVFRAG, which receives the message and prints it on the console screen. To run this example two commands must be typed:

First, on the host in slot five, type:

```
RCVFRAG <CR>
```

Second, on the host in slot one, type:

```
SFRAG <CR>
```

The host terminal in Slot one will display:

```
This is a reply to a fragmented message.
```

The host terminal in Slot five will display:

```
This is the second fragment.
```

```
$title('rcvfrag - receive a fragmented message')
$compact

/****************************************************************************
 *                INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *           This software is supplied under the terms of a
 *           license agreement or nondisclosure agreement with
 *           Intel Corporation and may not be copied or disclosed
 *           except in accordance with the terms of that agreement.
 *
 *                     Copyright Intel Corporation 1987, 1988
 *                           All rights reserved
 *
 *   For Intel customers licensed for the iRMX II Operating
 *   System under an Intel Software License Agreement, this source code and
 *   object code derived therefrom are licensed for use on a single central
 *   processing unit for internal use only.  Necessary backup copies and
 *   multiple users are permitted.  Object Code derived from this source code
 *   is a Class I software product under the Intel Software License Agreement
 *   and is subject to the terms and conditions of that agreement.
 *
 *   For the right to make incorporations, or to transfer this software to
 *   third parties, contact Intel corporation.
 *
 */
/***************************************************************************/

/*********************************************************************
 *
 * MODULE NAME: rcvfrag
 *
 * DESCRIPTION: Receive a fragmented message and print the message contained
 *              at the beginning of each fragment.
 *
 *
 *              The task receives a printable message.  If there is not enough
 *              buffer space to receive the entire message, receive the message
 *              in fragments.  (The info data structure associated with the
 *              rq$receive call contains the message length and the transaction
 *              id necessary to receive the message in fragments.)  Print
 *              the message and send a printable message to the sender.
 *
 *********************************************************************/
```

Figure 12-17. Receive a Message in Fragments (continued)

```
rcvfrag: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(dcom.ext)
$include(dcom.lit)
$include(/rmx286/inc/error.lit)
$include(err.ext)

DECLARE         /* Literals */

    FRAGLEN     LITERALLY   '1024',     /* fragmentation buffer length */
    BUFFRAG     LITERALLY   '0',        /* fragment is buffer flag */
    TSTPORT     LITERALLY   '801H',     /* well-known port */
    SFLAGS      LITERALLY   '00000B',   /* data buffer, synchronous flags*/
    NOEXCEPT    LITERALLY   '0';        /* no exception handling by system */


DECLARE         /* Global vars */

        status              WORD,
        port_t              TOKEN,    /* Token for local port */
        info                rec_info, /* info block on message received */
        buf_pool            TOKEN,    /* buffer pool attached to port */
        mes_buf(*)          BYTE initial (41,'This is a reply to a fragmented
                                           message',0dh,0ah),
        tran_id             WORD,     /* transaction id */
        bytes_rec           WORD,     /* number of bytes received in mess fragments */
        con_buf (20)        BYTE,     /* control message buffer */
        frag_buf(FRAGLEN)   BYTE,     /* fragmentation buffer */
        msg_ptr             POINTER;  /* pointer to received message */


    CALL set$exception(NOEXCEPT);
    port_t = get$dport(TSTPORT, @buf_pool, NOCHAIN, @status);
    msg_ptr = rq$receive(port_t, WAITFOREVER, @info, @status);
    CALL error$check(100, status);
    IF info.status = E$OK THEN DO; /* message may not be fragmented */
        CALL rqc$send$eo$response(NIL,0,msg_ptr,@status);
        CALL error$check(120, status);
        tran_id = rq$send$reply(port_t, info.rem$socket,
                                info.trans$id, @con_buf,
                                @mes_buf, size(mes_buf), SFLAGS, @status);
```

Figure 12-17. Receive a Message in Fragments (continued)

```
        CALL error$check(130, status);
        IF msg_ptr <> NIL THEN DO;
          CALL rq$release$buffer(buf_pool, SELECTOR$OF(msg_ptr), (info.flags AND 3),
                                      @status);
            CALL error$check(140, status);
        END;
    END;

    ELSE DO;
        IF info.status = E$NO$LOCAL$BUFFER THEN DO;

            /* receive fragments and print message at beginning of fragments */
            bytes_rec = 0;
            DO WHILE bytes_rec < info.data$length;
                CALL rq$receive$fragment(port_t, info.rem$socket, info.trans$id,
                                         @frag_buf, FRAGLEN, BUFFRAG, @status);
                CALL error$check(150, status);
                bytes_rec = bytes_rec + FRAGLEN;
                CALL rqc$send$eo$response(NIL,0,@frag_buf,@status);
                CALL error$check(160, status);
            END;

            /* complete transaction by sending a printable message */
            tran_id = rq$send$reply(port_t, info.rem$socket,
                                    info.trans$id, @con_buf,
                                    @mes_buf, size(mes_buf), SFLAGS, @status);
            CALL error$check(170, status);
        END;
        ELSE
            CALL rq$exit$io$job(0,NIL,@status);
    END;

    CALL rq$exit$io$job(0,NIL,@status);
 END rcvfrag;
```

Figure 12-17. Receive a Message in Fragments

```
$title('sfrag - initiate a transaction that forces receive fragmentation')
$compact

/****************************************************************************
 *            INTEL CORPORATION PROPRIETARY INFORMATION
 *
 *          This software is supplied under the terms of a
 *          license agreement or nondisclosure agreement with
 *          Intel Corporation and may not be copied or disclosed
 *          except in accordance with the terms of that agreement.
 *
 *                    Copyright Intel Corporation 1987, 1988
 *                          All rights reserved
 *
 *   For Intel customers licensed for the iRMX II Operating
 *   System under an Intel Software License Agreement, this source code and
 *   object code derived therefrom are licensed for use on a single central
 *   processing unit for internal use only.  Necessary backup copies and
 *   multiple users are permitted.  Object Code derived from this source code
 *   is a Class I software product under the Intel Software License Agreement
 *   and is subject to the terms and conditions of that agreement.
 *
 *   For the right to make incorporations, or to transfer this software to
 *   third parties, contact Intel corporation.
 *
 ****************************************************************************/

/*************************************************************************
 *
 *  MODULE NAME: sfrag
 *
 *  DESCRIPTION: Send a transaction request to a well-known socket.
 *               The request is sent with a data part to force receive
 *               fragmentation if the buffers at the receive port are
 *               less than 2K. Wait for a response and print the message
 *               on the console.
 *
 *************************************************************************/

sfrag: DO;

$include(/rmx286/inc/rmxplm.ext)
$include(dcom.ext)
$include (dcom.lit)
$include(/rmx286/inc/error.lit)
$include(err.ext)
```

Figure 12-18. Sending a Message that Requires Receive Fragmentation (continued)

```
DECLARE       /* Literals */

    REMPORT      LITERALLY   '801H',     /* Port id of remote port */
    REMHOSTID    LITERALLY   '5',        /* hostid of remote host */
    CONBUF       LITERALLY   '20',       /* control buffer size */
    RSVPB        LITERALLY   '128',      /* rsvp buffer size */
    TSTPORT      LITERALLY   '801H',     /* well-known port */
    NOEXCEPT     LITERALLY   '0',        /* no exception handling by system */
    SFLAGS       LITERALLY   '00000B';   /* data buffer, synch, rcvreply flags*/

DECLARE       /* Global vars */

    status              WORD,
    port_t              TOKEN,                  /* Token for local port */
    messock             socket,                 /* socket to which message is sent */
    msock               DWORD AT (@messock),    /* dword alias for messock */
    con_buf (CONBUF)    BYTE,                   /* control buffer */
    rsvp_buf (RSVPB)    BYTE,                   /* rsvp buffer */
    mess_size           DWORD,                  /* number of bytes in data message */
    rsvp_size           DWORD,                  /* rsvp buffer size */
    rsvp_ptr            POINTER,                /* points to rsvp message */
    info                rec_info,               /* receive info block */
    buf_pool            TOKEN,                  /* buffer pool attached to port */
    mbuf(2048)          BYTE INITIAL(37,'this was received via fragmentation',0ah,0dh),
    trans_id            WORD;                   /* transaction id */

    CALL set$exception(NOEXCEPT);
    port_t = get$dport(TSTPORT, @buf_pool, CHAIN, @status);
    messock.host_id = REMHOSTID;
    messock.port_id = REMPORT;
    mess_size = size(mbuf);
    rsvp_size = RSVPB;
    CALL MOVB(@(29,'This is the second fragment',0dh,0ah),@mbuf(1024), 30);
    trans_id = rq$send$rsvp(port_t,msock, @con_buf, @mbuf,
                                mess_size, @rsvp_buf, rsvp_size, SFLAGS, @status);
    CALL error$check(100, status);
    rsvp_ptr = rq$receive$reply(port_t, trans_id, WAITFOREVER, @info, @status);
    CALL error$check(110, status);
    CALL rqc$send$eo$response(NIL,0,@rsvp_buf,@status);
    CALL error$check(120, status);
    call rq$exit$io$job(0,NIL,@status);

END sfrag;
```

Figure 12-18. Sending a Message that Requires Receive Fragmentation

```
DECLARE
   socket       LITERALLY 'STRUCTURE(
                                 host_id     WORD,
                                 port_id     WORD)',

   host_info    LITERALLY 'STRUCTURE(
                                         th_count    WORD,
                                         next_id     WORD,
                                         hcount      WORD,
                                         res(2)      BYTE,
                                         hostids(10) WORD)',

   port_info_s  LITERALLY   'STRUCTURE(
                                         port_id     WORD,
                                         type        BYTE,
                                         reserved    BYTE,
                                         flags       WORD)',

   rec_info     LITERALLY   'STRUCTURE(
                                         flags       WORD,
                                         status      WORD,
                                         trans$id    WORD,
                                         data$length DWORD,
                                         for$port    TOKEN,
                                         rem$socket  DWORD,
                                         con$msg(20) BYTE,
                                         reserved(4) BYTE)',

    blk_struc        LITERALLY 'STRUCTURE(
                                              b_size      WORD,
                                              buf_ptr     POINTER,
                                              res         WORD)',

   c_buf           LITERALLY 'STRUCTURE(
                                              size       WORD,
                                              buf(1)     BYTE)';
DECLARE                 /* constant literals */
   DATACHAIN    LITERALLY        '0001B',  /* data chain message flag */
   NODATACHAIN  LITERALLY            '0',  /* contiguous buffer mess flag */
      CHAIN        LITERALLY     '010B',  /* data chain buf pool creation flag */
      NOCHAIN      LITERALLY        '0',  /* no data chain buf pool creation flag *
   SYNCHTRANS   LITERALLY            '0',  /* synchronous transmission flag */
   ASYNCHTRANS  LITERALLY      '010000B',  /* asynchronous transmission flag */
   RECRES       LITERALLY'0100000000B',  /* receive used for send$rsvp */
   RECREPLY     LITERALLY            '0',  /* receive$reply used for send$rsvp */
   NOTRAN       LITERALLY  '000000000B',  /* transactionless message */
   STATMESS     LITERALLY  '000010000B',  /* status message */
   TREQUEST     LITERALLY  '000100000B',  /* transaction request message */
   TRESPONSE    LITERALLY  '001000000B';  /* transaction response mess */
```

Figure 12-19. Literal File DCOM.LIT

```
create$buf$pool: PROCEDURE(max_bufs, init_num_bufs, attrs, status_ptr) TOKEN
EXTERNAL;

   DECLARE          /* Parameters */

       max_bufs      WORD,      /* maximum number of buffers in buffer pool */
       init_num_bufs WORD,      /* initial number of buffers in pool */
       attrs         WORD,      /* buffer pool creation attributes */
       status_ptr    POINTER;   /* exception pointer */

END create$buf$pool;

get$dport: PROCEDURE(port_id,buf_pool_ptr,b_attrs,status_ptr) TOKEN EXTERNAL;

   DECLARE     /* Parameters */

       port_id       WORD,
       buf_pool_ptr  POINTER,
       b_attrs       WORD,
       status_ptr    POINTER;

END get$dport;
```

Figure 12-20. External File DCOM.EXT

12.5.6.7 The Name Server Example

This is the most complex example provided to the user with the Extended iRMX II Operating System. This example implements a table that is used to dynamically catalog the names of all the ports created in a system. Two tasks, one for remote requests and one for local requests, manage the name server table.

The remote server task uses both control and data messages to service requests. The local server services requests through data mailboxes. Both tasks are needed because the Nucleus Communication Service cannot be used for local communication. The name server table itself is implemented as a circular list which is accessed by a group of procedures that insert or delete port names, get or change socket information, and set up the table for these accesses.

When a client board makes a request to the name server, the request is sent, the calling task waits for a reply, and the name server returns information specific to the request (e.g., the result of modifying an entry in the table, or the socket for a remote port.)

In order to run the name server example, the following commands are necessary:

```
ATTACHFILE /RMX286/DEMO/PLM/MB2/NSERVR <CR>
```

This command makes the directory containing the name server example the current directory. Next, type:

```
SUBMIT COMPILE <CR>
```

This command invokes a Command Sequence Definition (CSD) file that generates the executable name server and all of its required modules.

The name server can be run as a background job one of the processors. To start the name server running as a background job type:

```
BACKGROUND NSERVR > NSERVR.DOC <CR>
```

See the *Extended iRMX II Operator's Guide To The Human Interface* manual for information on the background command.

Two modules are provided which demonstrate the use of the name server. NSSNDMSG and NSRCVMSG which execute as a pair. NSRCVMSG must execute first, it posts a socket with the nameserver under the name "receiver." NSSNDMSG then executes, sending the nameserver a look-up request on the name "receiver." NSSNDMSG then sends a message to "receiver"; NSRCVMSG prints the message "This is a simple message", to the terminal console.

This process can be demonstrated on either host board, but the order of module execution cannot be changed.

12.6 GLOSSARY

Agent--Any board that is connected to the MULTIBUS II parallel system bus (iPSB).

bus interface--The Message Passing Coprocessor (MPC) chip is sometimes referred to as the bus interface. The purpose of the MPC is to provide a transparent interface between the local CPU and the parallel system bus (iPSB).

Client--A physical board (usually a processor board) that requests a service from another board. During a read from a disk, the processor board that requests the read is the client. See also Server.

data chain--A method of receiving data messages that are larger than any one buffer can hold. Data chaining is performed transparently by the system hardware.

datagram--The message format used by MULTIBUS II. Datagrams can be described as similar to mailing a letter. You write a letter, address it, put a stamp on it a place it in a mailbox. You assume that the letter will get to its destination, or if a reply is needed, you put an RSVP in the letter itself.

Interconnect Space--A group of 512 registers that contain information about each board. The primary use of this space is to replace physical jumpers. The configuration of a board can be changed by writing to interconnect space rather than inserting or removing physical jumpers.

message--All data and interrupts sent over the MULTIBUS II Parallel System Bus (iPSB). Messages can be thought of as a block of bytes sent over the bus that contains all of the information needed to send the message to the intended destination agent (board) and receive a reply, if requested. Two types of messages are supported in MULTIBUS II, solicited and unsolicited. See also unsolicited messages and solicited messages.

port--A data structure defined in the transport protocol that is used in passing messages. It provides a level of addressing that permits sending data to a particular task (program) running on a board.

Server--A physical board (frequently a controller that provides data storage) that provides a service to another board. During a read from a disk, the controller board that does the read and sends the data to the requestor is the server. See also Client.

solicited messages--Any data message that requires negotiation for buffer resources. Solicited messages are used to send data, such as disk read and write data, from one board to another. (See also, message and unsolicited message.)

source/destination address--An eight-bit field in every message passed on the parallel system bus (iPSB). This eight-bit field allows the unsolicited message to act as a virtual interrupt, this addressing scheme permits a total of 255 possible interrupts or boards, in a single system.

unsolicited messages--Any message that comes over the iPSB bus that was not requested by the receiving agent (board). Unsolicited messages are used as interrupts, or control signals. They relieve the local CPU from having to poll for messages coming over the bus. The unsolicited message is 32 bytes long. (See also, message, solicited message, and source/destination address.)

virtual interrupt--A software-routed interrupt that is contained in a MULTIBUS II message. Each MULTIBUS II message contains an eight-bit field that specifies the source and destination of the message. These source and destination bits allow the message to act as an interrupt to the Message Passing Coprocessor (MPC).

CHAPTER 13
NUCLEUS CONFIGURATION

13.1 INTRODUCTION

The Nucleus is a configurable part of the operating system. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, Intel provides three kinds of information:

- A list of configurable options
- Detailed information about the options
- Procedures to help you specify your choices

The rest of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the *iRMX II Interactive Configuration Utility Reference Manual.*

13.2 HARDWARE

The operating system supports a variety of hardware environments. By using the ICU, you can tailor the operating system to match your hardware. In particular, you can specify information about the following hardware elements:

Timer	You can specify the timer's base port, interval between ports, clock interrupt level, and clock frequency.
NPX	You can specify the addition of a Numeric Processor Extension for tasks requiring floating point instructions. The default option assumes that no NPX is present. If there is an NPX in your system, but you do not indicate it during configuration, your application cannot use NPX instructions. If you specify an NPX during configuration and your system does not contain an NPX, you may cause unexpected results.

13.3 SYSTEM CHARACTERISTICS

When you configure the Nucleus, you can specify a number of characteristics that affect your system:

Parameter Validation	A system call validates input parameters by checking for the existence of objects and by verifying that the objects are the correct type. If your system does not include the Basic I/O System, you can exclude parameter validation from your system.
GDT Entries	Each iRMX II object requires one GDT entry. Therefore, you need to configure the number of GDT slots your system requires.
IDT Entries	You can allocate the number of IDT entries, up to 256, that your system needs in the interrupt descriptor table.
Default Exception Handler	You can choose from one of four options for your system default handler: • Use the system default exception handler that deletes offending tasks. • Use the alternative system exception handler that suspends rather than deletes. • Use the iRMX II System Debugger as the exception handler. • Supply your own exception handler.
Round Robin Scheduling	You can determine if round robin scheduling will be in effect. If so, you can set the priority below which tasks will be assigned round robin scheduling, and the number of clock ticks each task may run before being rescheduled.

13.4 SYSTEM INITIALIZATION ERROR REPORTING

During the configuration process, you can elect to have initialization errors reported for each layer of the operating system. This is done by configuring Initialization Error Reporting (RIE) into your system when you configure the Nucleus. Then, whenever the operating system encounters an initialization error in a layer, it displays the following message and relinquishes control to the monitor:

```
<layer name> Initialization Error:  <error code number>
```

If Initialization Error Reporting is not configured into the Nucleus and an initialization error occurs, a code indicating the layer responsible for the initialization error and the corresponding error code are placed in the first two words of the Nucleus data segment (1E0:0000H). The Nucleus initialization task then goes into an infinite loop.

The codes for the layers that can cause an initialization error are

1 = Nucleus failure

2 = BIOS failure

3 = EIOS failure

4 = Human Interface failure

APPENDIX A EXTENDED iRMX II DATA TYPES

The Extended iRMX II Operating System recognizes these data types:

BYTE	An unsigned, eight-bit, binary number.
WORD	An unsigned, two-byte, binary number.
DWORD	An unsigned, 32-bit binary number, occupying two contiguous words of memory.
INTEGER	A signed, two-byte, binary number stored in two's complement form.
POINTER	Two words containing the segment selector and an offset, (offset first).
SELECTOR	A 16-bit quantity that is equivalent to the selector portion of a POINTER.
TOKEN	A word containing the logical address of an object. Tokens are selectors that reference an entry in a descriptor table. The entry in the descriptor table contains the physical address of the object.
STRING	A sequence of consecutive bytes having this structure:

```
length  BYTE,
chars (255) BYTE;
```

The first byte contains the length of the string (the number of succeeding bytes).

The subscript of the chars field (255) is the maximum number of bytes in any string. Note, that some system calls limit strings to lengths shorter than 255 bytes.

APPENDIX B OBJECT TYPES AND RESOURCE REQUIREMENTS

B.1 INTRODUCTION

This appendix lists the type codes for all iRMX II objects. In addition, it documents the amount of memory needed to create Basic I/O System objects.

B.2 OBJECT TYPES

Each iRMX II object type is known within iRMX II systems by means of a numeric code. Table B-1 lists the types with their codes.

Table B-1. Type Codes

OBJECT TYPE	NUMERIC CODE
Job	1
Task	2
Mailbox	3
Semaphore	4
Region	5
Segment	6
Extension	7
Composite	8
User	100
Connection	101
I/O Job	300
Logical Device	301
User-Created Composite	varies from 8000H to 0FFFFH depending on the value specified in CREATE$EXTENSION

NOTE: Users and connections are described in the *Extended iRMX II Basic I/O System User's Guide in Volume 2.* I/O jobs and logical devices are described in the *Extended iRMX II Extended I/O System Reference Manual.*

B.3 RESOURCE REQUIREMENTS

The Basic I/O System obtains memory from the calling job's memory pool when creating objects. The values listed here reflect Release 3 of the iRMX II Operating System.

Object	Number of 16-byte paragraphs required by the Basic I/O System
I/O Result Segment	4 (5 for an internal IORS that the Operating System creates when attaching a device)
Connection (to named file)	6
Connection (to physical file)	4
User object	3 (minimum)

Intel®

APPENDIX C EXCEPTION CODES

C.1 INTRODUCTION

Table C-1 provides a complete list of the Extended iRMX II condition codes that can occur during system operations. It lists the condition codes by layer with their numeric values and mnemonics.

Table C-1. Conditions and Their Codes

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
E$OK	The most recent system call was successful.	0H	0
	Nucleus Environmental Conditions		
E$TIME	A time limit (possibly a limit of zero time) expired without a task's request being satisfied.	1H	1
E$MEM	There is not sufficient memory available to satisfy a task's request.	2H	2
E$BUSY	Another task currently has access to the data protected by a region.	3H	3
E$LIMIT	A task attempted an operation which, if it had been successful, would have violated a Nucleus-enforced limit.	4H	4
E$CONTEXT	A system call was issued out of context or the operating system was asked to perform an impossible operation.	5H	5
E$EXIST	A token parameter has a value which is not the token of an existing object.	6H	6

--continued--

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Numeric Code Decimal
E$STATE	A task attempted an operation which would have caused an impossible transition of a task's state.	7H	7
E$NOT$CON-FIGURED	This system call is not part of the present configuration.	8H	8
E$INTER-RUPT$SAT-URATION	An interrupt task has accumulated the maximum allowable number of SIGNAL$-INTERRUPT requests.	9H	9
E$INTER-RUPT$OV-ERFLOW	An interrupt task has accumulated more than the maximum allowable amount of SIGNAL$INTERRUPT requests.	0AH	10
E$TRANS-MISSION	A NACK, timeout, or bus error occurred.	0BH	11
E$SLOT	There are no available GDT slots.	0CH	12
E$DATA$CHAIN	A data chain has been returned. The token points to a data chain block.	0DH	13
Nucleus Communications System Environmental Conditions			
E$CANCELLED	A SEND$RSVP transaction has been remotely cancelled.	00E1H	225
E$HOST$ID	The host$id portion of the socket parameter is not valid.	00E2H	226
E$NO$LOCAL$-BUFFER	The local buffer is too small to hold the message data.	00E3H	227
E$NO$REMOTE$-BUFFER	The buffer on the remote agent is too small to hold the message data.	00E4H	228
E$RESOURCE$-LIMIT	Either the simultaneous messages, or or transactions is not adequate.	00E6H	230

continued

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Numeric Code Decimal
E$TRANS$ID	The transmission is already done, or the specified trans$id is invalid.	00E8H	232
E$DISCON-NECTED	The socket is zero and the local port is not connected.	00E9H	233
E$TRANS$LIMIT	There has been a transmission resource limitation.	00EAH	234
	I/O System Environmental Conditions		
E$FEXIST	The specified file already exists.	20H	32
E$FNEXIST	The specified file does not exist.	21H	33
E$DEVFD	The device driver and file driver are not compatible.	22H	34
E$SUPPORT	The combination of parameters entered is not supported.	23H	35
E$EMPTY$-	The specified entry in a directory file is empty.	24H	36
E$DIR$END	The specified directory entry index is beyond the end of the directory file.	25H	37
E$FACCESS	The connection does not have the correct access to the file.	26H	38
E$FTYPE	The requested operation is not valid for this file type.	27H	39
E$SHARE	The requested operation attempted an improper kind of file sharing.	28H	40
E$SPACE	There is no space left on the volume.	29H	41
E$IDDR	An invalid device driver request occurred.	2AH	42
E$IO	An I/O error occurred.	2BH	43

continued

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
E$FLUSHING	The connection specified in the call was deleted before the operation completed.	2CH	44
E$ILLVOL	The device contains an invalid or improperly formatted volume.	2DH	45
E$DEV$OFF-LINE	The device being accessed is now offline.	2EH	46
E$IFDR	An invalid file driver request occurred.	2FH	47
E$FRAGMENT-ATION	The file is too fragmented to be extended.	30H	48
E$DIR$NOT$-EMPTY	The call is attempting to delete a directory that is not empty.	31H	49
E$NOT$FILE$-CONN	The connection parameter is a device connection, not a file connection.	32H	50
E$NOT$DEV-ICE$CONN	The connection parameter is not a device connection.	33H	51
E$CONN$NOT$-OPEN	The connection is not open for reading, writing or updating.	34H	52
E$CONN$OPEN	The task attempted to open a connection that is already open.	35H	53
E$BUFFERED$-CONN	The specified connection was opened by the EIOS, and used by the BIOS which is not allowed. Once you have an open connection, you must manipulate it with a system call provided by the same I/O System.	36H	54
E$OUTSTAND-ING$CONNS	A soft detach was specified, but connections to the device still exist.	37H	55

continued

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
E$ALREADY$-ATTACHED	The specified device is already attached.	38H	56
E$DEV$-DETACHING	The file specified is on a device that the operating system is in the process of detaching.	39H	57
E$NOT$SAME$-DEVICE	The existing pathname and the new pathname refer to different devices. You cannot simultaneously rename a file and move it to another device.	3AH	58
E$ILLOGICAL$-RENAME	The call is attempting to rename a directory to a new path containing itself.	3BH	59
E$STREAM$-SPECIAL	A stream file request is out of context. Either it is a query request and another query request is already queued, or it is a satisfy request and either the request queue is empty or a query request is queued.	3CH	60
E$INVALID$-FNODE	The connection refers to a file with an invalid fnode. You should delete this file.	3DH	61
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.	3EH	62
E$FNODE$-LIMIT	The volume already contains the maximum number of files. No more fnodes are available for new files.	3FH	63
E$LOG$NAME$-SYNTAX	The specified pathname starts with a colon (:), but it does not contain a second, matching colon; the specified pathname has more than 12 characters or contains invalid characters.	40H	64
E$IOMEM	The Basic I/O System has insufficient memory to process a request.	42H	66

continued

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
E$MEDIA	The device containing a specified file is not on-line.	44H	68
E$LOG$NAME$-NEXIST	The specified path contains an explicit logical name, but the Extended I/O System was unable to find the name in the object directories of the local job, the global job, and the root job.	45H	69
E$NOT$OWNER	The user who attempted to detach the device is not the owner of the device.	46H	70
E$IO$JOB	The Extended I/O System cannot create an I/O job because the size specified for the object directory is too small.	47H	71
E$UDF$FORMAT	The User Definition File is not in the right format.	48H	72
E$NAME$-NEXIST	The user name specified in the call is not listed in the User Definition File.	49H	73
E$UID$NEXIST	The user ID in the specified user object does not match the ID listed in the User Definition File for the corresponding user name.	4AH	74
E$PASSWORD-$MISMATCH	The password specified in the call does not match the one listed in the User Definition File for the corresponding user name.	4BH	75
E$UDF$IO	The User Definition File specified cannot be found.	4CH	76
E$IO$UNCLASS	An unknown type of I/O error occurred.	50H	80
E$IO$SOFT	A soft I/O error occurred. A retry might be successful.	51H	81

-continued-

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
E$IO$HARD	A hard I/O error occurred. A retry is probably useless.	52H	82
E$IO$OPRINT	The device was off-line. Operator intervention is required.	53H	83
E$IO$WRPROT	The volume is write-protected.	54H	84
E$IO$NO$DATA	A tape drive attempted to read the next record, but it found no data.	55H	85
E$IO$MODE	A tape drive attempted a read/write operation before the previous write (read) completed.	56H	86
E$IO$NO$-SPARES	An attempt was made to assign an alternate track, but no more alternate tracks were available.	57H	87
E$IO$ALT$-ASSIGNED	An alternate track was assigned during this I/O operation.	58H	88
	Application Loader Environmental Conditions		
E$BAD$HEADER	The object file contains an invalid header record.	62H	98
E$EOF	The Application Loader encountered an unexpected end-of-file while reading a record.	65H	101
E$NO$LOAD-ER$MEM	There is insufficient memory to satisfy the memory requirements of the Application Loader.	67H	103
E$NO$START	The Application Loader could not find the start address.	6CH	108
E$JOB$SIZE	The maximum memory-pool size of the job being loaded is smaller than the amount of memory required to load its object file.	6DH	109

continued

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
E$OVERLAY	The overlay name does not match any of the overlay module names.	6EH	110
E$LOADER$-SUPPORT	The file requires features not supported by the Application Loader as configured.	6FH	111
	Human Interface Environmental Conditions		
E$LITERAL	The parsing buffer contains a literal with no closing quote.	80H	128
E$STRING$-BUFFER	The string to be returned exceeds the size of the buffer the user provided in the call.	81H	129
E$SEPARATOR	The parsing buffer contains a command separator.	82H	130
E$CONTINUED	The parse buffer contains a continuation character.	83H	131
E$INVALID$-NUMERIC	A numeric value contains invalid characters.	84H	132
E$LIST	A value in the value list is missing.	85H	133
E$WILDCARD	A wild-card character appears in an invalid context, such as in an intermediate component of a pathname.	86H	134
E$PREPOSI-TION	The command line contains an invalid preposition.	87H	135
E$PATH	The command line contains an invalid pathname.	88H	136
E$CONTROL$C	The user typed a CONTROL-C to abort the command.	89H	137
E$CONTROL	The command line contains an invalid control.	8AH	138

continued

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Numeric Code Decimal
E$UNMATCHED-$LISTS	The number of files in the input and output pathname lists is not the same.	8BH	139
E$INVALID-$DATE	The operator entered an invalid date.	8CH	140
E$NO$PARAM-ETERS	A command expected parameters, but the operator didn't supply any.	8DH	141
E$VERSION	The Human Interface is not compatible with the version of the command the operator invoked.	8EH	142
E$GET$PATH-$ORDER	A command called C$GET$OUTPUT$-PATHNAME before calling C$GET$INPUT$PATHNAME.	8FH	143
E$PERMISSION	The user does not have permission to to access the requested resource.	90H	144
E$INVALID-$TIME	The operator entered an invalid time.	91H	145
	UDI Environmental Conditions		
E$UNKNOWN-$EXIT	The program exited normally.	0C0H	192
E$WARNING$-EXIT	The program issued warning messages.	0C1H	193
E$ERROR$EXIT	The program detected errors.	0C2H	194
E$FATAL$EXIT	A fatal error occurred in the program.	0C3H	195
E$ABORT$EXIT	The operating system aborted the program.	0C4H	196
E$UDI$INTER-NAL	A UDI internal error occurred.	0C5H	197

continued

Table C-1. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Numeric Code Decimal
	Nucleus Programmer Errors		
E$ZERO$-DIVIDE	A task attempted a divide in which the quotient was larger than 16 bits.	8000H	32768
E$OVERFLOW	An overflow interrupt occurred.	8001H	32769
E$TYPE	A token referred to an existing object that is not of the required type.	8002H	32770
E$PARAM	A parameter that is neither a token nor an offset has an invalid value.	8004H	32772
E$BAD$CALL	An OS extension received an invalid function code.	8005H	32773
E$ARRAY$-BOUND	Hardware or software has detected an array overflow.	8006H	32774
E$NDP$ERROR	A Numeric Processor (NPX) error has occurred. OS extensions can return the status of the NPX to the exception handler.	8007H	32775
E$ILLEGAL$-OPCODE	The processor tried to execute an invalid instruction.	8008H	32776
E$EMULATOR$-TRAP	An ESC instruction was encountered with the emulator bit set in the machine status word.	8009H	32777
E$CHECK$EX-CEPTION	A PASCAL task has exceeded the bounds of a CASE statement.	800AH	32778
NDP$SEGMENT-$OVERRUN	The NPX tried to access an address that is out of segment boundaries.	800BH	32779
E$PROTECTION	A general protection error.	800DH	32781
E$NOT$-PRESENT	A request has been made to load a a segment register whose segment is not present.	800EH	32782

continued

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

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
E$BAD$ADDR	The logical address is illegal- either the selector does not point to a valid segment or the offset is not within the segment boundaries.	800FH	32783
	Nucleus Communications Programmer Errors		
E$PROTOCOL	The port specified is of the signal type, not the data communication type.	80E0H	32992
E$PORT$ID$- USED	The specified port$id is already in use.	80E1H	32993
E$NUC$BAD$BUF	The buffer referred to is invalid, or not large enough.	80E2H	32994
	I/O System Programmer Errors		
E$NOUSER	No default user is defined.	8021H	32801
E$NOPREFIX	No default prefix is defined.	8022H	32802
E$BAD$BUFF	Illegal usage of memory buffers in read or write requests.	8023H	32803
E$NOT$LOG$- NAME	The specified object is not a device connection or file connection.	8040H	32832
E$NOT$DEVICE	A token parameter referred to an existing object that is not, but should be, a device connection.	8041H	32833
E$NOT$CON- NECTION	A token parameter referred to an existing object that is not, but should be, a file connection.	8042H	32834
	Application Loader Programmer Error		
E$JOB$PARAM	The maximum memory pool size specified for the job is less than the minimum pool size specified.	8060H	32864

continued

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

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
	Human Interface Programmer Errors		
E$PARSE$-TABLES	There is an error in the internal parse tables.	8080H	32896
E$JOB$TABLES	An internal Human Interface table was overwritten, causing it to contain an invalid value.	8081H	32897
E$DEFAULT$SO	The default output name string is invalid.	8083H	32899
E$STRING	The pathname to be returned exceeds 255 characters in length.	8084H	32900
E$ERROR$-OUTPUT	The command invoked by C$SEND$-COMMAND includes a call to C$SEND$-EO$RESPONSE, but the command connection does not permit C$SEND$EO$-RESPONSE calls.	8085H	32901
	UDI Programmer Errors		
E$RESERVE$-PARAM	The calling program tried to reserve memory for more than 12 files or buffers.	80C6H	32966
E$OPEN$PARAM	The calling program requested more than two buffers when opening a file.	80C7H	32967

INDEX

A

access rights
    discussion of 6-1
aliases 7-1
allocating memory 5-3
analogy of how MULTIBUS II systems work 12-2
applications 2-1
assigning levels to each interrupt 9-5

b

BITBUS interconnect 12-3
buffer pools 12-10
buffer pools 5-4
buffer pools, system calls for
    CREATE$BUFFER$POOL 12-11
    CREATE$BUFFER$POOL, DELETE$BUFFER$POOL 12-14
    REQUEST$BUFFER 12-11
    REQUEST$BUFFER, RELEASE$BUFFER 12-15
Built-in Self Tests (BIST) 12-6

C

call gates 1-7
Call-gates and OS extensions 1-3
case sensitivity of object directory names 6-2
child job 2-1
comparison of procedures, tasks, and OS extensions 10-2
composite objects
    deleting 11-3
    system calls for 11-2
composite objects 1-2

condition codes
  defined 8-1
  literal file 8-1
  ranges 8-1
  types 8-1
configuration of the Nucleus, see chapter 121
context switches 3-2
controlling memory pool size 5-2
creating an OS extension 1-2
creating new operating system objects 11-1

D

data chain block 12-12
data chains 12-11
data type, see Appendix A1
declaring your own exception handler 8-2
default time quota for round-robin scheduling 3-5
definition of
  round-robin scheduling 3-4
deleted of nested composites 11-5
descriptor 1-6
descriptor tables
  global 1-7
  interrupt 1-7
  local 1-7
descriptors
  aliases 7-2
  cautions on using 7-2
  changing the address or length of 7-2
  creating 7-2
  defining memory with 7-2
  definition of 7-1
  deleting 7-2
  explicit 7-1
  system calls for 7-2
  type code returned for 7-1
directory, object 1-4
disabling interrupts 9-6
discussion of round-robin scheduling 3-5

E

enabling interrupts 9-21
entering an object's name in the object directory 6-3
examples
    interrupt servicing 9-24
    ring buffer manager 11-7
exception handlers
    inherited 8-2
    invoking 8-3
exception handlers 8-2
exception handling
    for 80286 processor 8-4
    in-line 8-4
exception mode 8-3
exceptional conditions
    defined 8-1
    environmental 8-1
    programmer errors 8-1
exchange types 4-1
execution states of tasks 3-1
extension objects 1-2

F

features of MULTIBUS II systems 12-1
four types of address space
    I/O 12-3
    interconnect 12-3
    memory 12-3
    message 12-3
Free Space Manager 5-1

G

getting an object's name 6-2
getting an object's type code 6-2
global clock, MULTIBUS II 12-5

H

handlers
    exceptional 1-8
    interrupt 1-8
handling spurious interrupts 9-22

I

in-service register, examining 9-23
Interconnect address space 12-5
interconnect space, calls for
  GET$INTERCONNECT 12-13
  SET$INTERCONNECT 12-13
interrupt controllers 9-3
interrupt descriptor table 9-4
interrupt handler, using 9-11
interrupt handlers and tasks 9-9
interrupt levels 9-3
interrupt lines 9-3
interrupt mechanisms 9-2
interrupt servicing, using multiple buffers 9-16
interrupt task priorities 9-12
interrupt task, using 9-11
interrupts
  system calls for 9-27
interrupts 9 or 16 8-6
interrupts, handling spurious 9-22
interrupts, limit on outstanding 9-19
iSBX I/O expansion bus 12-3

J

job 2-1
job deletion 2-2
job tree 2-1

L

limit on outstanding interrupts 9-19

M

mailbox
  discussion of queues 4-2
  mechanics 4-1
  queues 4-1
mailboxes
  system calls for 4-3

mailboxes 4-1
memory
  how to allocate 5-3
  returning to the system 5-3
memory allocation 1-5
memory allocation performance feature 5-4
memory management
  system calls 5-4
memory needed to create an object, see Appendix B1
memory pool 1-4
memory pools 5-2
message agents, calls for getting information
  GET$HOST$ID 12-14
Message Fragmentation 12-13
message space, calls for
  BROADCAST 12-13
  CANCEL 12-14
  RECEIVE 12-14
  RECEIVE$FRAGMENT 12-14
  RECEIVE$REPLY 12-14
  SEND 12-13
  SEND$REPLY 12-13
  SEND$RSVP 12-13
Message Space Calls that Support the MIC Device 12-14
movement of memory between jobs 5-2
MULTIBUS II hardware overview 12-3
MULTIBUS II systems, an analogy of how they work 12-2
MULTIBUS II systems, features of 12-1
MULTICHANNEL DMS I/O bus 12-3
multiple buffers and interrupt servicing 9-16

N

NUC$ERROR 1-8
Nucleus Communication Service 12-8
number of interrupts possible on cascaded PICs 9-3
number of interrupts possible on one PIC 9-3
numeric codes for object types, see Appendix B1

O

object access 6-2
object directories
  using 6-2

object directory 1-4
  entering an object's name 6-3
object queues
  high-performance portion 4-2
object types 1-1
objects
  extension and composite 1-6
  job 1-3
  mailbox 1-5
  region 1-6
  segment 1-5
  semaphore 1-5
  system calls to manipulate 6-3
  tasks 1-2
objects 1-1
OS extension interface procedures 10-7
OS extensions
  and customized exception codes 10-14
  and error procedures 10-8
  entry procedures 10-7
  exception handling in 10-11
  function procedure 10-8
  functions of the interface procedures 10-7
  including into your system 10-14
  linking the procedures 10-14
  making objects immune from deletion 10-15
  procedures needed 1-2
  system calls for 10-15

P

parallel system bus (iPSB) 12-3
parameter object 2-2
parent job 2-1
port 12-9
ports, information provided on creation 12-9
port,system calls for
  ATTACH$BUFFER$POOL 12-10
  ATTACH$PORT 12-10
  ATTACH$PORT, DETACH$PORT 12-15
  CONNECT 12-10, 15
  CREATE$PORT, DELETE$PORT 12-15
  GET$PORT$ATTRIBUTES 12-10, 15
priority for tasks 3-1
procedures needed in an OS extension 1-2

R

regions
  cautionary notes 4-10
  deadlock and 4-9
  discussion of sharing data 4-6
  mutual exclusion 4-7
  system calls for 4-10
regions 4-5
relationships between interrupt tasks and handlers 9-15
resource sharing 2-1
restrictions when assigning interrupt levels 9-6
returning memory to the system 5-3
round-robin scheduling 3-4
RQ$ERROR 10-8

S

segments 5-1
semaphores4
  mutual exclusion 4-6
  system calls for 4-5
  task queue 4-4
semaphores 4-4
send$message
  acknowledging 4-2
serial system bus(iSSB) 12-3
services provided by the Central Services Module 12-4
setting up an interrupt handler 9-10
spurious interrupts, using GET$LEVEL to detect 9-23
system calls
  for interrupts 9-27
  type manager 11-18
system calls 1-2
system calls for
  tasks 3-6
system calls for exception handlers 8-6
system calls for memory management 5-4
system calls that manipulate the 80286 processor's access byte 6-1
system calls to manipulate jobs 2-2
system calls to manipulate objects 6-3
system initialization error reporting 12-3

T

task attributes 3-6
task execution states 3-1
task priority 3-1
task resources 3-6
task state transition 3-2
task states
  asleep 3-1
  asleep-suspended 3-2
  ready 3-2
  running 3-2
  suspended 3-2
tasks 3-1
three ways to add functionality in iRMX 10-1
token 1-2
tools for interrupts 9-2
Transport Protocol 12-8
two system calls that set up the interrupt descriptor table 9-6
type manager
  jobs during DELETE$EXTENSION 11-5
  jobs during DELETE$JOB 1-3
typical actions of an exception handler 8-2

U

use of PL/M-286 DISABLE statement in tasks with interrupts diabled 9-7
using an interrupt handler 9-11
using an interrupt task 9-11

EXTENDED iRMX®II BASIC I/O SYSTEM USER'S GUIDE

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

Intel®

PREFACE

INTRODUCTION

This manual documents the Basic I/O System, one of the layers of the iRMX II Operating System. The material contained herein is intended primarily as introductory and background information for using the system calls. You can find detailed information for using these system calls in the *Extended iRMX II Basic I/O System Calls Reference Manual.*

Readers who are familiar with the iRMX I Basic I/O System will also be familiar with the iRMX II version. The iRMX II Operating System is based on the iRMX I Operating System.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus Use's Guide* and with the PL/M-286 programming language.

MANUAL ORGANIZATION

This manual is divided into eight chapters. Some of the chapters contain introductory or overview material that you do not need to read if you are already familiar with the iRMX II subsystems. Other chapters contain reference material that you will refer to as you write your application tasks. You can use this chapter to determine which of the other chapters you need to read. The manual organization is as follows:

Chapter 1	This chapter describes the features of the Basic I/O System. You should read this chapter if you are going through the manual for the first time or if you have had very little previous exposure to the Basic I/O System.
Chapter 2	This chapter explains some basic terminology associated with the Basic I/O System, including the concepts of system programmer, device, volume, file, and connection. You should read this chapter if you are looking through the manual for the first time or if you are unfamiliar with the Basic I/O System.

Chapter 3-5	These chapters describe named, physical, and stream files and how to use them. You should read one or more of these chapters, depending on the kinds of files your application uses. The use of remote files is not described in this manual.
Chapter 6	This chapter describes general information about synchronous and asynchronous system calls.
Chapter 7	This chapter lists the configuration options that pertain to the Basic I/O System.

CONVENTIONS

This manual uses the following conventions:

- The term "iRMX II" refers to the Extended iRMX II.3 Operating System.
- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- All iRMX II system calls begin with one of two standard prefixes: RQ$ or RQE$. When referring to the system calls that begin with RQ$, this manual uses a shorthand notation and omits the prefix. For example, S$CREATE$FILE means RQ$S$CREATE$FILE. The actual PL/M-286 external procedure names used to invoke these system calls are shown only in the *Extended iRMX II Extended I/O System Calls Reference Manual*, which lists the detailed calling sequences.
- All iRMX II system calls begin with one of two standard prefixes: RQ$ or RQE$. When referring to the system calls that begin with RQ$, this manual uses a shorthand notation and omits the prefix. For example, A$CREATE$FILE means RQ$A$CREATE$FILE. The actual PL/M-286 external procedure names used to invoke these system calls are shown only in the *Extended iRMX II Basic I/O System Calls Reference Manual*, which lists the detailed calling sequences.
- When referring to system calls that begin with RQE$, this manual spells out the complete names, including the RQE$ characters.
- You can also invoke the system calls from assembly language, but to do so you must obey the PL/M-286 calling conventions that are discussed in the *Extended iRMX II Programming Techniques Manual*.

CONTENTS

CHAPTER 1 **PAGE**
FEATURES OF THE BASIC I/O SYSTEM

1.1 Introduction 1-1
1.2 16M-byte Memory Addressability 1-2
1.3 Protection Features 1-2
1.4 Synchronous and Asynchronous Operation 1-2
1.5 Device Independence 1-3
1.6 Support for Many Kinds of Devices 1-3
1.7 Four Distinct Kinds of Files 1-4
1.7.1 Named Files 1-4
1.7.2 Physical Files 1-4
1.7.3 Stream Files 1-4
1.7.4 Remote Files 1-5
1.8 File Sharing and Access Control 1-5
1.8.1 File Sharing 1-5
1.8.2 Access Control 1-5
1.9 Separation of File Lookup and File Open Operations 1-5
1.10 Control Over Internal Fragmentation of Files 1-6
1.11 Global Time-of-Day Clock 1-7
1.12 Disk Integrity 1-7
1.12.1 Attach Flags 1-8
1.12.2 Fnode Checksum Field 1-8
1.12.3 Getting and Setting the Bad Track/Sector Information 1-8

CHAPTER 2 **PAGE**
FUNDAMENTAL CONCEPTS

2.1 Introduction 2-1
2.2 System Programmers 2-1
2.3 Device Controllers and Device Units 2-1
2.4 Volumes 2-2
2.5 Files 2-2
2.6 Connections for Task and Device-Unit Communication 2-2
2.6.1 Interlayer Bonds Preceding Initialization 2-3
2.6.2 Post-Initialization Bond - the Configuration Interface 2-4
2.6.2.1 Device Connections 2-4
2.6.2.2 File Connections 2-5
2.6.2.3 Some Observations about Devices and Connections 2-8

CHAPTER 3 PAGE
NAMED FILES
3.1 Introduction 3-1
3.2 Multiple Files on a Single Device 3-1
3.3 Hierarchical Naming of Files 3-1
3.3.1 Connections 3-3
3.3.2 Paths 3-4
3.3.3 Prefix and Subpath 3-4
3.3.4 Default Prefix 3-4
3.4 Controlling Access to Files 3-5
3.4.1 Users and User Objects 3-5
3.4.1.1 User IDs 3-6
3.4.1.2 User Objects 3-6
3.4.1.3 Default User Object For a Job 3-7
3.4.2 Types of Access to Files 3-7
3.4.3 File Access List 3-8
3.4.4 Computing Access for File Connections 3-9
3.4.5 Special Users 3-11
3.4.5.1 System Manager User 3-11
3.4.5.2 World User 3-11
3.4.6 Example 3-11
3.5 Extension Data 3-13
3.6 System Calls for Named Files 3-13
3.6.1 Obtaining and Deleting Connections 3-13
3.6.2 User Objects 3-14
3.6.3 Default Prefixes 3-15
3.6.4 Manipulating Data 3-15
3.6.5 Obtaining Status 3-17
3.6.6 Reading Directory Entries 3-17
3.6.7 Deleting and Renaming Files 3-17
3.6.8 Changing Access 3-18
3.6.9 Identifying a File's Name 3-18
3.6.10 Manipulating Extension Data 3-18
3.6.11 Detecting Changes in Device Status 3-18
3.7 Accessing the Global Time-of-Day Clock 3-19
3.8 Accessing Files Through iRMX-NET 3-19
3.9 Chronological Overview of Named FIles 3-19
3.9.1 Most Frequently Used System Calls 3-19
3.9.2 Calls Relating to User Objects 3-19
3.9.3 Calls Relating to Prefixes 3-20
3.9.4 Calls Relating to Status 3-20
3.9.5 Calls Relating to Changing Access 3-20

CHAPTER 3 (continued) **PAGE**

3.9.6 Calls for Monitoring Device Readiness 3-21
3.9.7 Calls Relating to Extension Data 3-21
3.9.8 Calls for Renaming Files 3-21
3.9.9 Calls for Identifying File Names 3-21
3.9.10 Calls for iRMX-NET 3-21

CHAPTER 4 **PAGE**
PHYSICAL FILES

4.1 Introduction 4-1
4.2 Situations Requiring Physical Files 4-1
4.3 Connections and Physical Files 4-2
4.4 Using Physical Files 4-2

CHAPTER 5 **PAGE**
STREAM FILES

5.1 Introduction 5-1
5.2 Actions Required of the Writing Task 5-1
5.3 Actions Required of the Reading Task 5-3

CHAPTER 6 **PAGE**
SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

6.1 Introduction 6-1
6.2 Synchronous System Calls 6-1
6.2.1 User Parameter 6-2
6.2.2 File-Path Parameter(s) for Named Files 6-2
6.2.3 Response Mailbox Parameter 6-4
6.2.4 I/O Buffers 6-6
6.3 Asynchronous System Calls 6-6
6.4 Condition Codes 6-10

CHAPTER 7 **PAGE**
CONFIGURING THE BASIC I/O SYSTEM

7.1 Introduction 7-1
7.2 Basic I/O System Calls 7-1
7.3 Intel I/O Devices 7-1
7.4 Buffers 7-1
7.5 Timing Facilities 7-2
7.6 Service Task Priorities 7-2
7.7 Creating a File with an Existing Pathname 7-2
7.8 System Manager 7-2
7.9 System Initialization Error Reporting 7-2
7.10 Factors Affecting Basic I/O System Performance 7-3

APPENDIX A **PAGE**
iRMX® II DATA TYPES

A.1 Data Types A-1

APPENDIX B PAGE
OBJECT TYPES AND RESOURCE REQUIREMENTS
B.1 Introduction B-1
B.2 Object Types B-1
B.3 Resource Requirements B-2

APPENDIX C PAGE
I/O RESULT SEGMENT
C.1 Overview C-1
C.2 Structure of I/O Result Segment C-2
C.3 Unit Status For Specific Devices C-3
C.3.1 iSBC® 214/215G Controller C-3

APPENDIX D PAGE
EXCEPTION CODES
D.1 Overview D-1
D.2 Sequential (Environmental) Exception Codes D-1
D.3 Sequential (Programmer Error) Exception Codes D-2
D.4 Concurrent (Environmental) Exception Codes D-2
D.5 Concurrent (Programmer Error) Exception Code D-3

APPENDIX E PAGE
LOGICAL DEVICES AND THE BASIC I/O SYSTEM
E.1 Logical Devices E-1
E.2 References E-1

APPENDIX F PAGE
iRMX® I AND iRMX® III BASIC I/O SYSTEM DIFFERENCES
F.1 Introduction F-1
F.2 80286 Capabilities F-1
F.2.1 16M-Byte Addressability F-1
F.2.2 Memory Protection F-2
F.3 Device Drivers F-2
F.4 Disk Integrity Features F-2
F.4.1 Attach Flags F-2
F.4.2 Fnode Checksum Field F-3
F.4.3 Getting and Setting the Bad Track/Sector Information F-3

TABLES

TABLE PAGE

B-1 Type Codes B-1

FIGURES

FIGURE PAGE

2-1 Layers of Interfacing Between Tasks and a Device 2-3
2-2 Schematic of Software at Initialization Time 2-4
2-3 A System with Device and File Connections 2-6
3-1 Example of a Named-File Tree 3-2
3-2 Computing the Access Mask for a File Connection 3-10
3-3 Chronology of Frequently Used System Calls for Named Files 3-20
6-1 Sample Named File Tree 6-5
6-2 Concurrent Behavior of an Asynchronous System Call 6-8

CHAPTER 1
FEATURES OF THE BASIC I/O SYSTEM

1.1 INTRODUCTION

Because the iRMX II Operating System is designed for use by Original Equipment Manufacturers (OEMs), it provides a large number of features--including some that are not generally found in operating systems aimed at end users. These features include

- 16M-byte memory addressability
- Memory protection
- Synchronous and asynchronous system calls
- Device independence
- Support for many kinds of devices
- Four distinct kinds of files
- File sharing and access control
- Separation of file lookup and file open operations
- Control over fragmentation of files
- Global time-of-day clock
- Disk integrity

This chapter explains each of these features and familiarizes you with the terminology of the Basic I/O System.

NOTE

All material on iRMX Networking Software (iRMX-NET) can be found in the *iRMX Networking Software User's Guide*. This manual is not part of the iRMX II manual set.

1.2 16M-BYTE MEMORY ADDRESSABILITY

The iRMX II Operating System runs in Protected Virtual Address Mode (PVAM) of the 80286 or 80386 processors. As a result, it can access as much as 16M bytes of memory. The Basic I/O System takes advantage of this feature by allowing you to create I/O jobs with memory pools of up to 16M bytes. Therefore, tasks that invoke Basic I/O System calls can have more code and can have more room for data than with iRMX I.

Application tasks must use logical addresses to access memory. Logical addresses take the form:

```
selector:offset
```

Some device controllers also support a 16M-byte address space. These controllers use physical addresses (direct 24-bit addresses) to refer to the memory space. If you write your own device drivers for these controllers, your device drivers must know how to convert logical addresses to physical addresses. The *Extended iRMX II Device Drivers User's Guide* discusses this technique.

1.3 PROTECTION FEATURES

Because the iRMX II Operating System accesses the processor in PVAM, it benefits from some of the inherent memory protection features of the processor. These features protect your code and data by preventing any task from reading or writing buffers of memory unless it has explicit access to those buffers. They also prevent memory reads or writes from crossing segment boundaries. The Operating System generates exception codes if an attempted protection violation occurs.

The Operating System also checks system call parameters for protection violations and for incorrect values. Appendix D lists the exception codes that can be returned.

1.4 SYNCHRONOUS AND ASYNCHRONOUS OPERATION

When you examine the *Extended iRMX II Basic I/O System Calls Reference Manual*, you will find that the system calls can be divided into two categories according to their names. The first category consists of system calls having names of the form:

```
RQ$XXXXX
```

where XXXXX is a brief description of what the system call does. The second category consists of system calls having names of the form:

```
RQ$A$XXXXX
```

System calls of the first category, without the A, are synchronous calls. They begin running as soon as your application invokes them, and continue running until they detect an error or accomplish everything they must do. Then they return control to your application. In other words, synchronous calls act like subroutines.

System calls of the second category (those with the A) are called asynchronous because they accomplish their objectives by using tasks that run concurrently with your application. This allows your application to accomplish some work while the Basic I/O System deals with devices such as disks or tape drives.

1.5 DEVICE INDEPENDENCE

The Basic I/O System provides you with one set of system calls that can be used with any collection of devices. For instance, rather than using a TYPE system call for output to a terminal and a PRINT system call for output to a line printer, you can use a WRITE system call for output to any device.

This notion of one set of system calls for I/O to any collection of devices is called device independence, and it allows your application much flexibility. For example, suppose that your application logs events as they occur. The device independence of the Basic I/O System allows you to create an application that can log the events on any device rather than just one.

For instance, when the event application is running and circumstances force an operator to reroute logging from the teletypewriter to the line printer, your application can be written to do this.

For a more detailed explanation of device independence, refer to the *Introduction to the Extended iRMX II Operating System*.

1.6 SUPPORT FOR MANY KINDS OF DEVICES

Although your application can be device independent, the Basic I/O System must be able to communicate with a wide variety of devices. To connect a particular device to the Basic I/O System, you must have a device driver (a collection of software procedures) designed especially for the device being connected.

The Basic I/O System provides device drivers for many devices. The *Extended iRMX II Interactive Configuration Utility Reference Manual* lists these devices and describes how to include their device drivers in your application system. If you need drivers for other devices, you must supply the drivers. Refer to the *Extended iRMX II Device Drivers User's Guide* for instructions on how to write your own device driver.

1.7 FOUR DISTINCT KINDS OF FILES

Files in the Basic I/O System are byte-oriented (as opposed to record-oriented files). The System provides you with four kinds of files: named, physical, stream, and remote.

1.7.1 Named Files

Named files are intended for use with random-access, secondary storage devices such as disks, diskettes, and bubble memories. They allow your application to organize its files into a tree-like, hierarchical structure that reflects the relationships between the files and the application. Furthermore, only named files allow your application to store more than one file on a device, and only named files provide your application with access control. Named files also provide a good starting place for building custom access methods such as the indexed sequential access method (ISAM).

For more information regarding named files, refer to Chapter 4.

1.7.2 Physical Files

Physical files differ from named files in that each physical file occupies an entire device. In fact, from the standpoint of the Basic I/O System, a physical file is a device. Yet with the Basic I/O System, an application can deal with a physical file as if it were a string of bytes.

Physical files provide several important advantages:

- An application can have direct control over a device.
- This direct control provides complete flexibility. For example, an application can interpret volumes created by other systems.
- An application can conserve memory and still be able to communicate with devices that do not need the power of named files. Examples of such devices include line printers, display tubes, plotters, and robots.

The disadvantages of physical files, as compared to named files, are that hierarchical file structures and access control are not available.

1.7.3 Stream Files

Stream files provide a means of intertask communication. Some tasks can write into a stream file while other tasks read from it concurrently. Stream files use no devices and provide no access control. They are implemented in memory.

1.7.4 Remote Files

The Basic I/O System can also access remote files through iRMX-NET. For more information on accessing remote files, consult the *iRMX Networking Software User's Guide*.

1.8 FILE SHARING AND ACCESS CONTROL

The Basic I/O System provides your application with the ability to share files and, in the case of named files, to control access to the files.

1.8.1 File Sharing

In a multitasking system, it may be useful to have several tasks manipulating a file simultaneously. Consider, for example, a transaction processing system in which a large number of operators concurrently manipulate a common data base. If each terminal is driven by a distinct task, the only way to implement an efficient transaction system is to have the tasks share access to the data base file. The iRMX II Operating System allows multiple tasks to concurrently access the same file.

1.8.2 Access Control

Also useful in a multitasking system is the ability to control access to a file. For instance, suppose that several engineering departments share a computer. An engineer in one department may want to reserve for himself the ability to delete his files, while allowing people in his department to write and read his file, and people in other departments to only read the files. The Basic I/O system named files provide your applications with this kind of access control.

1.9 SEPARATION OF FILE LOOKUP AND FILE OPEN OPERATIONS

Many operating systems waste valuable time by looking up a file whenever an application tries to open one. The Basic I/O System avoids this by using a special type of object (called a connection) to represent the bond between the file and a program.

Whenever your application software creates a file, the Basic I/O System returns a connection. Your application can then use the connection to open the file without suffering the expense of having the Basic I/O System lookup the file. Even when your application wants to open an existing file, the application can present the connection and bypass the file lookup process.

There are several other benefits associated with connection objects. In the case of named files, connections embody access rights to the file. This means that access need only be computed once (when the connection is created) rather than each time the file is opened.

A second benefit of connections is that several connections can simultaneously exist for the same file. This allows several tasks to concurrently access different locations in the file. This is possible because each connection maintains a file pointer to keep track of the location, within the file, where the task is reading or writing.

1.10 CONTROL OVER INTERNAL FRAGMENTATION OF FILES

When information is stored on a mass storage device, space is allocated in blocks rather than one byte at a time. These blocks are called granules, and the block size is called granularity. There are three kinds of granularity that are important.

device granularity

The device granularity is hardware dependent and varies among individual mass storage devices. It represents the minimum amount of data that the device can read or write during one I/O operation. For disk media, a device granule is called a sector; therefore the device granularity is the sector size. Each buffer that the Basic I/O System uses when reading and writing data is equal in size to the device granularity.

volume granularity

Volume granularity is a multiple of the device granularity. It represents the minimum amount of space that can be allocated to a file at one time. The Basic I/O System uses an algorithm based on volume granularity when deciding where on the volume to allocate this space. You specify the volume granularity when you format the volume.

file granularity

File granularity is a multiple of volume granularity which specifies the actual amount of space on the mass storage device that the Basic I/O System allocates to a file at one time. You assign the file granularity on a per-file basis when you invoke A$CREATE$FILE to create the files.

By selecting the proper granularity values, you can minimize fragmentation of your volumes. Use the following guidelines when selecting these values:

- Device granularity depends on hardware. If your device supports multiple device granularities, selecting the larger value usually gives higher performance. Although you can obtain greater performance, you may waste storage space due to a few large granules containing only a few bytes of data.
- For flexible diskettes, always set the volume granularity equal to the device granularity, unless you plan to store many large files on the volume. Even then, don't select a volume granularity larger than 1K (1024 bytes).

- For hard disks, set the volume granularity equal to the device granularity, unless the device granularity is less than 1K. Then set the volume granularity to 1K.
- When creating a large file, assign a large file granularity to minimize the number of noncontiguous blocks that make up the file. This decreases the fragmentation of the volume. For smaller files, set the file granularity equal to the volume granularity to minimize wasted space on the volume.

1.11 GLOBAL TIME-OF-DAY CLOCK

Some boards supported by the iRMX II Operating System have an on-board, battery backed-up, time-of-day clock. The Basic I/O System reads and writes the time of day, taking advantage of this clock feature. The global time-of-day clock is global in the sense that it is the timekeeper for the entire system. The iRMX II Operating System maintains a "local" time-of-day clock of its own, which is a copy of the global clock. The global and local clocks keep track of two items:

- The current date (day, month, and year)
- The current time (hours, minutes, and seconds)

The iRMX II Operating System needs two time-of-day clocks because it takes much longer (100 milliseconds and up) to access the global clock than the local clock. Therefore, the iRMX II Operating System maintains the local time-of-day clock for its date and time needs, and accesses the global time-of-day clock only during system initialization or upon request from the operator.

The iRMX II Basic I/O System provides two system calls that enable your applications to read the global date and time and set them to new values. These system calls are GET$GLOBAL$TIME and SET$GLOBAL$TIME. It also provides two system calls for manipulating the local clock: GET$TIME and SET$TIME.

1.12 DISK INTEGRITY

In any computer system, there are many occurrences beyond the control of the program or programmer that can cause damage to files or disk volumes. For example, power outages can occur just as a file is being written, or marginal disk sectors can suddenly become unreliable. The Basic I/O System has several features that enable programs to maintain disk integrity and determine whether files or volumes have been corrupted. The following sections outline these features.

1.12.1 Attach Flags

The Basic I/O System maintains flags that can indicate the integrity of named volumes and named files. Whenever you attach a named volume, the Basic I/O System sets a flag in the volume label to indicate that the volume is attached. Likewise, when you attach a named file, the Basic I/O System sets a flag in the fnode (file descriptor node) file to indicate that the file is attached. When you detach a volume or file, the Basic I/O System clears the associated flag, indicating that the file or volume was successfully detached.

Although the Basic I/O System doesn't check these flags to determine file or volume integrity, you can check the condition of a volume by invoking the A$GET$FILE$STATUS system call.

The Basic I/O System doesn't provide a system call for checking the file flag. However, you can write your own programs to check this flag, or you can use the Disk Verification Utility to examine the fnode file.

1.12.2 Fnode Checksum Field

The Basic I/O System uses the fnode file to keep track of every named file on a volume. The fnode file lists such information as the file name, the creation and last modification dates, and the location of every disk sector that makes up the file. Whenever you access a file, the Basic I/O System uses the fnode file to determine the file's location on the volume. Whenever you create, modify, or delete a file, the Basic I/O System modifies the fnode file to match the changes you made.

When the last connection to the file is deleted, the Basic I/O System writes to the fnode file, and it always calculates a checksum and writes that value in one of the fields of the fnode file. This checksum can be used to determine whether any data errors occurred when the Basic I/O System wrote the fnode file. Although the Basic I/O System doesn't calculate another checksum and compare it against the original when it next reads the file, your programs can use the checksum field to determine whether the fnode file has become corrupted. DISKVERIFY and SHUTDOWN can be used.

1.12.3 Getting and Setting the Bad Track/Sector Information

It is not uncommon for a hard disk to have a few sectors or tracks that cannot reliably store information. Many of these disks have a record of these bad tracks written on the second-highest cylinder of the disk. When the Basic I/O System formats a disk, it uses this bad track/sector information to assign alternate tracks or sectors for the bad tracks/sectors listed. The A$SPECIAL system call also has the ability to retrieve and set the bad track/sector information on a volume. One subfunction allows you to retrieve the current list of defective tracks or sectors. Another subfunction enables you to set up a new bad track/sector list.

CHAPTER 2 FUNDAMENTAL CONCEPTS

2.1 INTRODUCTION

Before you use the Basic I/O System, you must understand several fundamental concepts. Some of these concepts were presented in Chapter 2. The remaining concepts are

- System programmers
- Device controllers and device units
- Volumes
- Files
- Connections

The following sections explain these concepts.

2.2 SYSTEM PROGRAMMERS

There are two programming roles associated with the iRMX II Operating System. One role involves using system calls and objects that affect only your own iRMX II job, while the other role involves manipulating system resources and characteristics. These two roles are called application programming and system programming.

Although the roles have different names, separate people are not required. One individual can perform both roles. The reason for the distinction is that the actions of the system programmer affect the performance and security of the entire system, whereas the actions of the application programmer have a more limited effect.

The *Extended iRMX II Basic I/O System Calls Reference Manual* gives you several system call descriptions that begin with caution notices. These system calls, if misused, can have serious consequences for an application system. Therefore, you should consider these system calls to be reserved for the exclusive use of system programmers.

2.3 DEVICE CONTROLLERS AND DEVICE UNITS

You are probably familiar with the notion of a device; a hardware entity that tasks can use to read or write information, or to do both. Devices include flexible diskette drives, line printers, terminals, card readers, and the like.

In the iRMX II environment, it is convenient to make a distinction between devices and the hardware interfaces that communicate directly with an iRMX II application system. A hardware entity that talks directly with iRMX II software is a device controller. Devices such as those named in the previous paragraph are device units. Typically, a device controller acts as an interface between iRMX II application software and several device units. For example, an iSBC 214 Winchester Controller board acts as an interface between application software and from one to four Winchester disk drives (device units.)

2.4 VOLUMES

A volume is the medium used to store the information on a device unit. For example, if the device unit is a flexible disk drive, the volume is a diskette; if the device unit is a bubble memory board, the volume is the bubble memory; and if the device unit is a multi-platter hard disk drive, the volume is the disk pack.

2.5 FILES

Some operating systems treat a file as a device, while others treat a file as information stored on a device. The Basic I/O System considers a file to be information.

The Basic I/O System supports four kinds of files, and each has characteristics that make it unique. Regardless of the kind of file, the Basic I/O System provides information to applications as a string of bytes, rather than as a collection of records.

2.6 CONNECTIONS FOR TASK AND DEVICE-UNIT COMMUNICATION

In complex environments such as those supported by the iRMX II Operating System, several layers of software and hardware must be bound together before communication between application tasks and device units can commence. Figure 2-1 shows these layers.

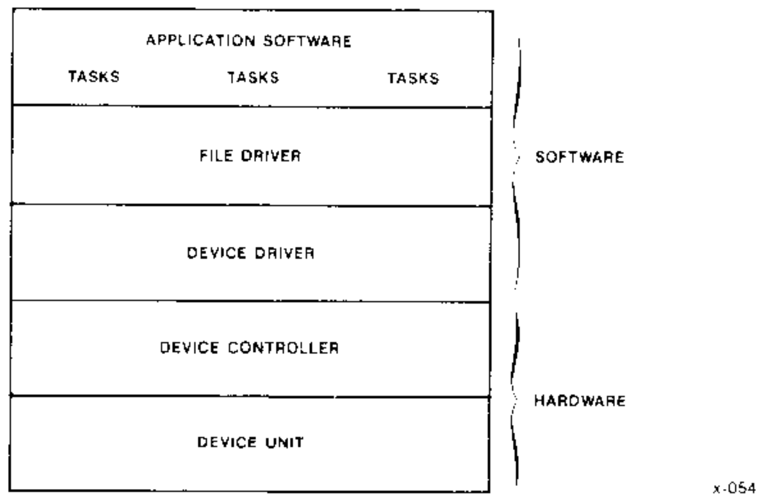


Figure 2-1. Layers of Interfacing Between Tasks and a Device

2.6.1 Interlayer Bonds Preceding Initialization

The bond between a device controller and the device units that it controls is a physical bond, usually in the form of wires or cables. A device driver is bound to device controllers by data residing in a data structure known as a Device Unit Information Block (DUIB). You supply the data for the DUIBs when you use the Interactive Configuration Utility (ICU) to configure the Operating System.

When your application starts up, there is a gap between the application software and the file drivers, and another gap between the file drivers and the device drivers. Figure 2-2 illustrates this situation. The new element, shown in the figure as the configuration interface, is the "glue" that provides the final bonds.

2.6.2 Post-Initialization Bond - the Configuration Interface

The configuration interface provides two kinds of system calls. Before a task can use a file, both of these kinds of calls must be invoked, and each produces a connection. These connections are called device connections and file connections, and several of them are shown in Figure 2-3 as conduits and wires through the conduits, respectively.

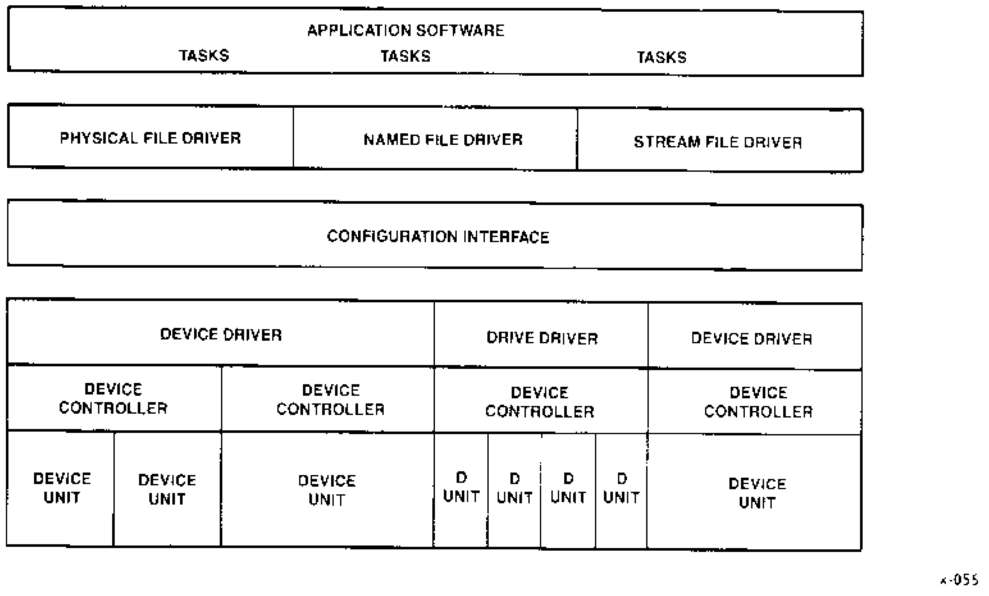


Figure 2-2. Schematic of Software at Initialization Time

2.6.2.1 Device Connections

Tasks employ the configuration interface first by calling the A$PHYSICAL$ATTACH$DEVICE system call, which returns a token for an iRMX II object type called a device connection. This device connection is the application's only pathway to the device. Moreover, there can be only one device connection between a device unit and all of the application tasks that need to use the device.

Because the device connection is so centrally important to the application, only tasks written by a system programmer should call A$PHYSICAL$ATTACH$DEVICE. Such a task could make the device connection available to application tasks selectively by sending it to certain mailboxes or by cataloging it in certain object directories. Or, to ensure that all required device connections will be available to all of the application tasks that need them, the system programmer could provide an initialization task that creates all of those device connections and catalogs them in the root object directory.

If and when the device is no longer needed by the application, an appropriate task can call A$PHYSICAL$DETACH$DEVICE to delete the device connection.

2.6.2.2 File Connections

When an application task is ready to use a device unit, it must use the device connection for that device unit to obtain a file connection object, which is a connection to a particular file on that device unit. How the task does this depends on whether the file already exists. If the file already exists, the task usually calls A$ATTACH$FILE, although it can also call A$CREATE$FILE. If the file does not yet exist, the task must call A$CREATE$FILE.

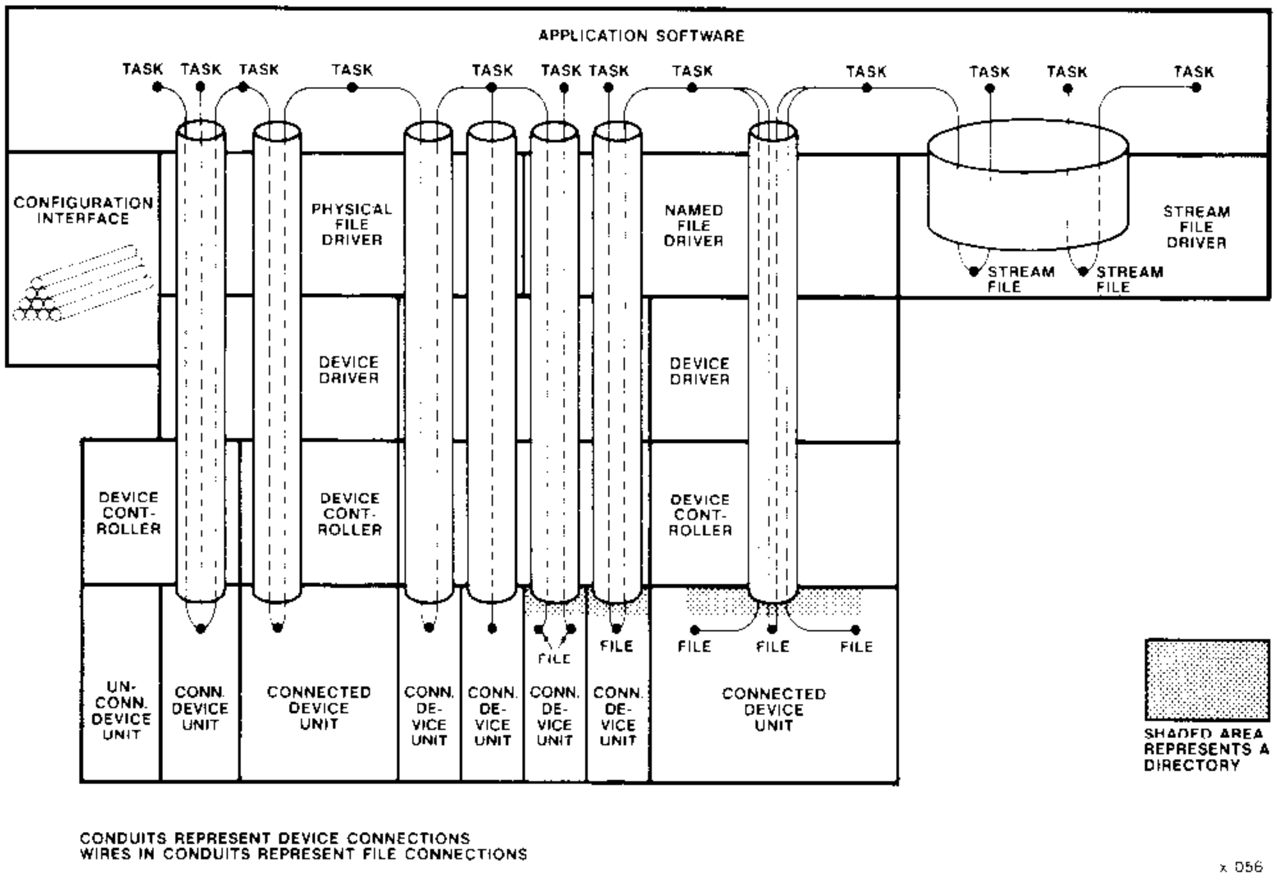


Figure 2-3. A System with Device and File Connections

NOTE

Even though a task can call A$CREATE$FILE to obtain a file connection for a file that already exists, it is not a good idea for a task to use A$CREATE$FILE unless the task is certain that the file does not yet exist. There are two reasons for this.

First, if a named file exists, then calling A$CREATE$FILE to obtain a connection to the file might cause the file to be truncated. This could cause problems for tasks having other connections to that file, because the file pointers (discussed later in this section) for those other connections are not affected, even though the end-of-file marker might be moved closer to the beginning of the file.

Second, if a file exists as either a physical or stream file, then it does not matter whether new connections to the file are obtained by a call to A$CREATE$FILE or A$ATTACH$FILE. However, it is possible that the code that does this will someday be used to create a connection to a named file, and as you can see, this can cause problems.

Unlike device connections, there can be multiple file connections to a single file. This allows different tasks, if necessary, to have different kinds of access to the same file at the same time, as the next paragraph shows.

After receiving a file connection, a task calls A$OPEN to open the connection. In the call to A$OPEN, the task specifies how it intends to use the file connection and how it is willing to share the file with other tasks using other connections, by passing the following as parameters:

- An open-mode indicator

  The open-mode indicator tells the Basic I/O System how your application is going to use the connection. This parameter can specify that the connection is open for reading only, for writing only, or for both reading and writing.

- A share-mode indicator

  The share-mode indicator specifies how other connections can share the file with the connection being opened. This parameter can specify that there can be no other open connections to the file, that other connections to the file can be opened for reading only, that other connections to the file can be opened for writing only, or that other connections to the file can be opened for both reading and writing.

For each open file connection to a random-access device unit, the Basic I/O System maintains a file pointer. This is a pointer that tells the Basic I/O System the logical address of the byte where the next I/O operation on the file is to begin. The logical addresses of the bytes in a file begin with zero and increase sequentially through the entire file. Normally the pointer for a file connection points at the next logical byte after the one most recently read or written. However, a task can use the file connection, if need be, to modify the file pointer by means of the A$SEEK system call.

2.6.2.3 Some Observations about Devices and Connections

Figure 2-3 is quite detailed and shows most of the situations that are possible for device units and file connections to them. In particular, you can observe the following:

- Device connections extend from the application software to the individual device units, and each passes through one and only one file driver.
- There is only one device connection to each connected device, and multiple file connections can share the same device connection.
- Different device units with the same controller can be connected via different file drivers.
- Tasks can share access to the same device unit through the physical file driver, and they can share access to the same files on the same device unit 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. There can be additional stream files in the application if more are needed.
- The configuration interface, which is depicted as a pile of conduits, is off to one 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.

CHAPTER 3
NAMED FILES

3.1 INTRODUCTION

Named files are intended for use with random-access, secondary storage devices such as disks, diskettes, and bubble memories. Named files provide several features that are not provided by physical or stream files. These features include

- Multiple Files on a Single Device
- Hierarchical Naming of Files
- Access Control
- Extra Data in a File's Descriptor

These features combine to make named files extremely useful in systems that support more than one application and in applications that require more than one file.

3.2 MULTIPLE FILES ON A SINGLE DEVICE

As shown in Figure 3-1, your application can use named files to implement more than one file on a single device. This can be very useful in applications requiring more than one operator, such as transaction processing systems.

3.3 HIERARCHICAL NAMING OF FILES

The iRMX II named files feature allows your application to organize its files into a number of tree-like structures as depicted in Figure 3-1. Each such structure, called a file tree, must be contained on a single device, and no two file trees can share a device. In other words, if a device contains any named files, the device contains exactly one file tree. Named file trees must also fit on a single volume.

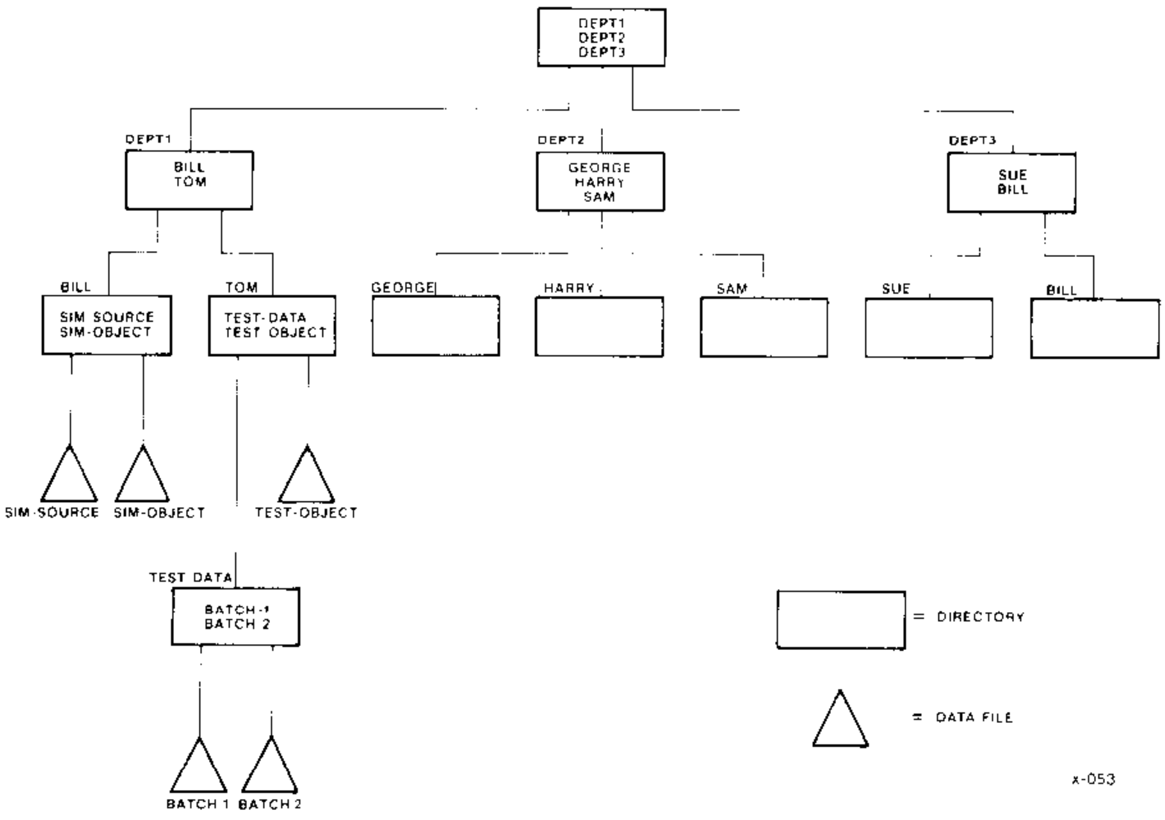


Figure 3-1. Example of a Named-File Tree

Each file tree consists of two categories of files--data files and directories. Data files (which are shown as triangles in Figure 3-1) contain the information that your application manipulates, such as inventories, accounts payable, transactions, text, source code, or object code. In contrast, directory files (shown as rectangles) contain only pointers to other files or directories. The purpose of the directory files is to provide you with flexibility in organizing your file structure.

To illustrate this flexibility, take a close look at Figure 3-1. This figure shows how named files can be useful in multi-user systems. Figure 3-1 is based on a collection of hypothetical engineers who work for three departments (Departments 1, 2 and 3). Each engineer is responsible for his own files. This multiperson organization is reflected in the file tree. The uppermost directory (called the device's root directory) points to three "department directories." Each department directory points to several "engineer's directories." And the engineers can organize their files as they wish by using their own directories.

Each file (directory or data) has a unique shortest path connecting it to the root directory of the device. For instance, in Figure 3-1, the file called SIM-SOURCE has the path DEPT1/BILL/SIM-SOURCE. This notion of "path" reflects the hierarchical nature of the named-file tree.

Another characteristic of hierarchical file naming is that there is less chance for duplicate file names. For example, note that Figure 3-1 contains directories for two individuals named Bill. (These directories are on the extreme left and right of the third level of the figure.) Even if the rightmost Bill had a data file with the file name of SIM-OBJECT, its path would differ from that leftmost Bill's SIM-OBJECT. Specifically, the leftmost SIM-OBJECT is identified by:

    DEPT1/BILL/SIM-OBJECT

whereas the rightmost SIM-OBJECT would be identified by:

    DEPT3/BILL/SIM-OBJECT

Whenever your application manipulates either kind of named file, the application must tell the Basic I/O System which file is to be manipulated. There are several ways to specify a particular named file to the Basic I/O System, all of which involve connections and paths.

3.3.1 Connections

Once you have a connection to a particular named file, you can use the connection as the PREFIX parameter of any system call. If, in the same call, you set the SUBPATH parameter to NIL or SELECTOR$OF(NIL), the Basic I/O System will ignore the SUBPATH and use only the PREFIX to find that particular file.

3.3.2 Paths

If you do not have a connection to the file, you can specify the file by using its path. To do this, build an iRMX II string. (An iRMX II string is a representation of a character string. To represent a string of n characters, you must use n + 1 consecutive bytes. The first byte contains the character count, n. The following n bytes contain the ASCII codes for the characters, in the same order as the string.) This string is called a path name. Then use a pointer to this path name as the SUBPATH parameter in the system call, and use the device connection as the PREFIX parameter in the system call.

For example, if your named file tree is on Drive 1, and it has the path name DEPT2/HARRY/TEST-RESULTS, you can specify the file by using the device connection for Drive 1 as the PREFIX parameter and a pointer to the path name as the SUBPATH parameter.

3.3.3 Prefix and Subpath

Once your application has obtained a connection to a directory file within a named file tree, the application can use that connection as a basis for reaching all files that descend from the directory.

For example, referring again to Figure 3-1, suppose your application has a connection to Directory DEPT1/TOM. The application can refer to Data File BATCH-1 by using both the PREFIX and the SUBPATH parameters. The application should use the connection to Directory DEPT1/TOM as the PREFIX, and it should use a pointer to a subpath name as the SUBPATH. The subpath name is a string that connects Directory DEPT1/TOM to Data File BATCH-1. For this example, the subpath name is TEST-DATA/BATCH-1.

3.3.4 Default Prefix

Within one iRMX II job, most references to a named file tree are generally confined to one branch of the tree. For example, in Figure 3-1, Tom will usually access the files in his directory more frequently than files outside of his directory. Recognizing this clustering, the Basic I/O System provides the notion of default prefix.

The Basic I/O System allows your application to specify one default prefix for each iRMX II job. A default prefix is a connection to a directory at the head of the most commonly used branch in your named file tree. For instance, in Figure 3-1, Tom's application would probably use a connection to Directory DEPT1/TOM as the default prefix. To use the default prefix, the application sets the PREFIX parameter to NIL or SELECTOR$OF(NIL).

A default prefix provides a job with two advantages. First, by providing a reference point within a named file tree, it allows your application to use subpath names instead of path names. If your tree is several levels deep, this can save programming time during development. Second, and more significantly, a default prefix provides a means of writing generalized application code that can work at any of several locations within a tree.

Consider an example. Suppose that an assembler (implemented as an iRMX II job) uses a default prefix to find a location in a named file tree. The assembler could then use a subpath name of TEMP to find or create a temporary work file. Before an application invokes the assembler, it sets the default prefix of the assembler job to a directory in the application's named file tree. This allows more than one job to invoke the assembler concurrently without the risk of sharing temporary files.

The Basic I/O System keeps track of a job's default prefix by using the job's object directory. Whenever your tasks use the SET$DEFAULT$PREFIX system call to specify a connection as being the default, the Basic I/O System catalogs the connection under the name $ in the job's object directory.

3.4 CONTROLLING ACCESS TO FILES

In most environments where files are shared among multiple users, it is necessary to have a means of controlling which users have access to which files. Among users who have access to a given file, it is frequently necessary to grant different kinds of access to different users. The iRMX II Operating System provides this control by identifying users with user IDs and embedding access rights for these IDs into the files. This section describes the user ID and file access mechanisms.

3.4.1 Users and User Objects

The iRMX II Operating System uses the concept of "user" to correlate file access to people or to iRMX II jobs. But the precise definition of "user" depends on the nature of your application.

If your application allows several people to enter information (at terminals, for example), you might want to consider each person (or small group of persons) a user. This allows each individual (or small group) to maintain access different from other individuals (or small groups).

Alternatively, if your application does not interact with people (or allows only one person to interact), you might wish to consider each iRMX II job as a user. This setup would allow your application to control the files that each job can access.

In more general terms, the set of entities that manipulate named files in your system is the set of all users. If you want all of these entities to be able to access any file, you can consider them to be a single user. However, if you want to distribute different access to different collections of these entities, you must divide the entities into subsets, each of which is a separate user.

For example, look at Figure 3-1. As mentioned earlier, all engineers are responsible for their own files. If engineers want to have unique access to their files (perhaps permitting no one else to use their files), each engineer must be a separate user. However, if all engineers are willing to give uniform access to other members of the department, then the department can be a separate user.

3.4.1.1 User IDs

A user ID is a 16-bit number that represents any individual or collection of individuals requiring a separate identity for the purpose of gaining access to files.

3.4.1.2 User Objects

The Basic I/O System uses a special type of object called a user object when determining access rights to files. A user object contains a list of one or more user IDs. When a task attempts to manipulate a file, it must supply the token for a user object. To determine access, the Operating System compares the IDs in the supplied user object with information contained in the file itself.

To understand user objects, consider an application in which every person who accesses the system is a separate user. In this situation, every person would have a separate user object. The user object represents the person.

The first ID in the user object is the owner ID. This is the ID of the user whom the object represents. If you think of a user object as a person, the owner ID represents the name of that person. When a person creates files, the Operating System automatically embeds the owner ID of that person's user object into the file, allowing that person automatic access to the file.

The IDs that follow the owner ID represent additional kinds of access that the person has. For example, people often belong to organizations such as athletic clubs and fraternal groups which distribute identity cards to their members. To participate in the organization, people must show their identity cards to prove they are members. The user IDs that follow the owner ID serve the same purpose. They identify the person as one of a select group, all of whom have the same access to a certain set of files.

The Basic I/O System has three system calls that manipulate user objects:

- CREATE$USER creates a user object and returns to the calling task a token for that user object.

- DELETE$USER deletes a user object.
- INSPECT$USER returns to the calling task the list of IDs in the user object specified in the call.

3.4.1.3 Default User Object For a Job

Most I/O operations performed within a particular iRMX II job are performed on behalf of one user object. Recognizing this, the Basic I/O System allows your application to designate a default user object for each job. Whenever your application invokes a Basic I/O System call on behalf of the default user object, the application can use SELECTOR$OF(NIL) as the token for the "user" parameter. The Basic I/O System recognizes the SELECTOR$OF(NIL) as referring to the default user.

The Basic I/O System provides two system calls to manipulate a job's default user. 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. GET$DEFAULT$USER can be used to ascertain the default user for a job.

The Basic I/O System uses the job's object directory to keep track of the job's default user object. Whenever one of your tasks sets or gets a default user object, the Basic I/O System either catalogs or looks up the entry for the default user object in the object directory. It uses the name R?IOUSER to refer to the default user object. To prevent problems, you should consider R?USER to be a reserved name, and you should avoid using it.

3.4.2 Types of Access to Files

Each of the two kinds of named files--directory files and data files--can be accessed in four different ways.

Every directory file can potentially be accessed in one or more of the following ways:

Delete	Delete the directory file with A$DELETE$FILE.
List	Obtain the contents of the directory file with A$READ or A$GET$DIRECTORY$ENTRY.
Add Entry	Add entries to the directory with A$CREATE$FILE, A$CREATE$DIRECTORY, or A$RENAME$FILE.
Change Entry	Change the access rights of files listed in the directory with A$CHANGE$ACCESS.

Every data file can potentially be accessed in one or more of the following ways:

Delete	Delete the file with A$DELETE$FILE or rename the file with A$RENAME$FILE.
Read	Read the file with A$READ.
Append	Add information to the end of the file with A$WRITE.
Update	Change information in the file with A$WRITE or drop information with A$TRUNCATE.

A user's access rights to a particular file depend on the access list associated with that file.

3.4.3 File Access List

For each named file (data or directory), the Basic I/O System maintains an access list which defines the users who have access and their access rights. Each access list is a collection of up to three ordered pairs having the form

ID, access mask

The ID portion is a user ID. The list of user IDs defines the users who can access the file.

The access mask portion defines the kind of file access that the corresponding user has. An access mask is a byte in which individual bits represent the various kinds of access permitted or denied that user. When such a bit is set to 1, it signifies that the associated kind of access is permitted. When set to 0, the bit signifies that the associated kind of access is denied.

The association between the bits of the access mask and the kinds of access they control are as follows (where bit 0 is the least-significant bit):

Bit	Directory Files	Data Files
0	Delete	Delete
1	List	Read
2	Add Entry	Append
3	Change Entry	Update

The remaining bits in the access mask have no significance.

For example, an access list for a data file might look like the following:

```
5B31  00001110
9F2C  00000010
```

where the ID numbers (left column) are in hexadecimal and the access masks (right column) are in binary. This means that the ID number 5B31 has read, append, and update access rights, while the ID number 9F2C has the read access right.

The first entry in the file's access list is placed there automatically by the Basic I/O System when it creates the file. The ID portion of that entry is the first ID number in the user object specified in the call to A$CREATE$FILE. That ID is known as the owner ID for the file. The access rights portion is supplied as a parameter in the same call.

Tasks can alter the access list of a file by means of the A$CHANGE$ACCESS system call. With A$CHANGE$ACCESS, you can add or delete ID-access pairs, and you can change the access rights of IDs already in the access list.

NOTE

The user whose ID is the owner ID for a file has one advantage over other users. Only a file's owner can use the A$CHANGE$ACCESS system call to modify the file's access list without being granted explicit permission to do so.

3.4.4 Computing Access for File Connections

Whenever a task calls A$CREATE$DIRECTORY, A$CREATE$FILE, or A$ATTACH$FILE, the Basic I/O System constructs an access mask and binds it to the file connection object returned by the call. This access mask is constant for the life of the connection, even if the access list for the file is subsequently altered. When the connection is used to manipulate the file, the access mask for the connection determines how the file can be accessed. For example, if the computed access rights for a connection to a data file do not include appending or updating, then that connection cannot be used in an invocation of A$WRITE.

When a task calls A$CREATE$DIRECTORY or A$CREATE$FILE, the access mask for the connection is the same as the access mask that the task supplies in the "access" parameter of the system call. However, when a task calls A$ATTACH$FILE, the Basic I/O System compares the user object specified in the "user" parameter with the file's access list and computes an aggregate mask.

Figure 3-2 illustrates the algorithm that the Basic I/O System uses during a call to A$ATTACH$FILE. As the figure shows, the Basic I/O System compares the IDs in the specified user object with the IDs in the file's access list. The access masks corresponding to matching IDs are logically ORed, forming an aggregate mask.

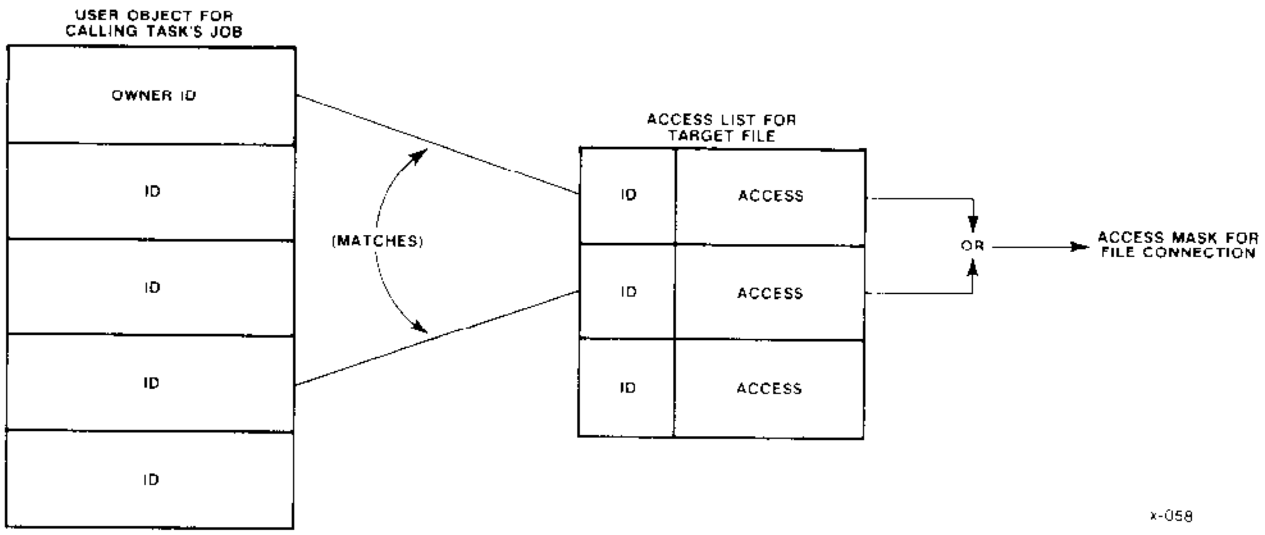

USER OBJECT FOR CALLING TASK'S JOB
OWNER ID
ID
ID
ID
ID
(MATCHES)
ACCESS LIST FOR TARGET FILE
ID
ACCESS
ID
ACCESS
ID
ACCESS
OR
ACCESS MASK FOR FILE CONNECTION
x-058

Figure 3-2. Computing the Access Mask for a File Connection

Normally, the Basic I/O System uses the aggregate access mask embedded in the connection to determine a task's ability to access a file. However, there are two circumstances in which the Basic I/O System computes access again: during A$CHANGE$ACCESS and during A$DELETE$FILE. When a task invokes one of these system calls, the Basic I/O System computes the access to the target file (or to the data file or directory specified in the "prefix" parameter, if the subpath portion is null). If the user object specified in the system call does not have appropriate access rights, the Basic I/O System denies the task the ability to delete the file or change the access.

NOTE

When computing access, the Basic I/O System checks the access only to the last file in the specified subpath and to the parent directory of the last file. It does not check the access to any other directory files specified in the path. If the subpath is null, the Basic I/O System checks the access to the file indicated by the "prefix" parameter.

3.4.5 Special Users

There are two user IDs that can have special meaning to the Basic I/O System. One is the number 0 (the system manager), which has special meaning to the Basic I/O System. The other is the number 0FFFFH (the WORLD user), which can have special meaning based on the application.

3.4.5.1 System Manager User

If so indicated during the configuration process, user ID 0 represents the "system manager." A user object containing this value is privileged in two respects. First, when it is used to create or attach files, the resulting file connection automatically has read access to data files and list access to directory files. This is true even if a file's access list does not contain an ID-access mask pair whose ID value is 0. The second privilege granted such a user object is that it can call A$CHANGE$ACCESS to change any file's access list.

3.4.5.2 World User

By convention, the user ID 0FFFFH represents WORLD (all users in the system). To implement this convention, you should place the ID for WORLD in the list of user IDs for every user object you create. This allows your application to set aside certain files as public files, giving everyone limited access. For example, your file system might contain a series of utilities, such as compilers or linkers, which all users need to access. Instead of granting everyone access on an individual basis (which is impossible if you have more than three users), you can grant the user WORLD access to the files. Since WORLD is on the ID list of every user object, this grants everyone access to the files.

As a side effect of including the WORLD ID in every user object, any file whose owner ID is 0FFFFH (WORLD) can have its access list modified by anyone. That is, any file connection for that file can be used in a call to A$CHANGE$ACCESS.

3.4.6 Example

The following example can help you understand how to use IDs, access masks, access lists, and user objects to permit each user in a system to have exactly the kinds of access that you want that user to have.

Referring back to Figure 3-1, suppose that Tom is to have all kinds of access to the file BATCH-1, that Bill is to have read and append access only, and that the members of Department 2 are to have read access only.

Tom (or whoever creates BATCH-1) can arrange for these kinds of access by doing the following:

- Create a number of user objects, one for Tom, one for Bill, and one for each of the members of Department 2 (George, Harry, and Sam). When creating the user objects, assign unique owner IDs for each user. Assume that the owner ID numbers are 4000H for Tom and 8000H for Bill. Assign unique owner IDs for each of the members of Department 2, but also include a common user ID (assume F000H for this example) as an additional ID in each of their user objects.)
- Use A$CREATE$FILE to create the file BATCH-1. In the call to A$CREATE$FILE, use the token for the user object containing the 4000H ID number and specify the access mask 00001111B. This call returns a file connection that gives its user (Tom) all kinds of access to BATCH-1. At this point, the access list for BATCH-1 has just one ID-access mask pair.
- Use A$CHANGE$ACCESS to add an ID-access mask pair to the access list of BATCH-1. The ID should be 8000H and the access mask should be 00000110B. This gives Bill read and append access to Batch-1. Now the access list for BATCH-1 has two ID-access mask pairs.
- Use A$CHANGE$ACCESS to add a third pair to the access list of BATCH-1. The ID should be F000H and the access mask should be 00000010B. This gives the people in Department 2 read access to BATCH-1.
- Inform Bill that he can read the contents of BATCH-1 and append new information to it. Describe to him the prefix and subpath that are needed to attach BATCH-1, and tell him to create a user object with the ID 8000H. Tell him to specify that user object when attaching BATCH-1.
- Inform the members of Department 2 that they can read the contents of BATCH-1. Describe for them the prefix and subpath needed to attach BATCH-1, and tell them to create user objects that contain an entry for the ID F000H. Tell them to specify those user objects when attaching BATCH-1.

When Bill attaches BATCH-1, he receives a file connection that he can use in calls to A$READ. He also can use A$WRITE, provided that the file pointer for that connection is at the end of the file.

When a member of Department 2 attaches BATCH-1, he receives a file connection that he can use in calls to A$READ.

Note that this example shows that one ID number can be used to give certain access rights to an individual and that another ID number can be used to give different access rights to a collection of individuals.

3.5 EXTENSION DATA

For each named file on a random access volume, the Basic I/O System creates and maintains a file descriptor on the same volume. The first portion of the descriptor contains information for the Basic I/O System. The last portion, called extension data, is available to your operating system extension. You specify the number (from 0 to 255, inclusive) of bytes of extension data for each named file on the volume, when formatting the volume with the FORMAT utility.

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

3.6 SYSTEM CALLS FOR NAMED FILES

Several system calls relate to iRMX II named files. Some of these calls are useful for both data and directory files, some for only one kind of file, and some (such as CREATE$USER) don't relate to either kind of file.

The following sections briefly explain the purpose of each of the system calls. The descriptions are grouped by function rather than alphabetically. These descriptions are very brief. The *Extended iRMX II Basic I/O System Calls Reference Manual* contains detailed descriptions of the calls.

3.6.1 Obtaining and Deleting Connections

Six system calls pertain to obtaining or deleting connections.

- A$CREATE$FILE

  This call applies only to data files. Your application must use this call to create a new data file, and it can use this call to obtain a connection to an existing data file. If the application uses this call to create a new file, the Basic I/O System automatically adds an entry in the parent directory for this new file.

- A$CREATE$DIRECTORY

  This call applies only to directory files. Your application must use this call to create a new directory file. The call cannot be used to obtain a connection to an existing directory. The Basic I/O System automatically adds an entry in the parent directory for this new directory.

- A$ATTACH$FILE

  This call applies to both data and directory files. Your application can use this call to obtain a connection to an existing data or directory file.

- A$DELETE$CONNECTION

  This call applies to both data and directory files. Your application can use this call to delete a connection to either kind of named file. This call cannot be used to delete a device connection.

- A$PHYSICAL$ATTACH$DEVICE

  This call does not directly apply to either data or directory files. Your application uses this call to obtain a connection to a device. Even though this connection is a device connection, it can be used as the prefix for the root directory of the device. However, using this system call causes a task to lose its device independence.

- A$PHYSICAL$DETACH$DEVICE

  This call does not directly apply to either data or directory files. Your application uses this call to delete a connection to a device.

3.6.2 User Objects

Five system calls pertain directly to user objects. None of these calls are specifically related to data or directory files. The calls are:

- CREATE$USER

  This call is used to create a user object.

- DELETE$USER

  This call is used to delete a user object.

- INSPECT$USER

  This call is used to ascertain a user object's id and to find out to which groups the user belongs.

- SET$DEFAULT$USER

  Your application can use this call to establish a default user for any iRMX II job.

- GET$DEFAULT$USER

  Your application can use this call to ascertain the default user for any iRMX II job.

3.6.3 Default Prefixes

Two calls pertain to default prefixes, and neither of these calls pertains directly to data files or directory files. The calls are:

- SET$DEFAULT$PREFIX

  Your application can use this call to set the default prefix for any iRMX II job.

- GET$DEFAULT$PREFIX

  Your application can use this call to ascertain the default prefix for any iRMX II job.

3.6.4 Manipulating Data

Eight system calls allow you to manipulate the data in a file. Four apply to both directory and data files, two apply to data files only, and two are auxiliary calls that aid in the data manipulation process. The system calls are:

- A$OPEN

  This call applies to both data and directory files. Before your application can use any other system calls to manipulate file data, the application must open a connection to the file. This system call is the only way to open a connection.

- A$CLOSE

  This call applies to both data and directory files. After your application has finished manipulating a file, the application can use this system call to close the file connection. Your application can elect to leave the file open, letting the Basic I/O System close it when the connection is deleted, but there is an advantage to closing connections when they are not being used.

  This advantage derives from the fact that, when a connection is shared between two or more applications, some of the applications can place restrictions on the manner of sharing. For instance, an application can specify sharing with writers only. By closing connections, your application can improve the likelihood that the connections can be used by other applications. A connection is not closed until all pending I/O requests have been handled.

- A$SEEK

  This system call applies to both data and directory files. Whenever your application reads, writes, or truncates a file, the application must tell the Basic I/O System the location in the file where the operation is to take place. To do this, your application uses the A$SEEK system call to position the file pointer of the file connection. The A$SEEK system call requires that the file connection be open.

- A$READ

  This system call applies to both data and directory files. Your application can use this system call to read file data from the location indicated by the file pointer and place the data in a memory buffer. Before using this system call, your application can use the A$SEEK system call to position the file pointer. The A$READ system call requires that the file connection be open. It also requires that the segment of memory to which you copy the data be a writable segment.

  The outcome of this system call depends upon whether a data file or a directory is being read. If your application reads a data file, the application will receive data that makes up the file. If the application reads from a directory, the application will receive data that represents the entries of the directory.

  Each entry in a directory consists of 16 bytes. The first two bytes contain a 16-bit file descriptor number corresponding to the file descriptor number associated with the A$GET$FILE$STATUS system call in the *Extended iRMX II Basic I/O System Calls Reference Manual.* The remaining 14 bytes are the ASCII characters making up the name of the file to which the directory entry points. (A file's name is the last component of a path name.) The advantage in using the A$READ system call to read a directory is that your application can obtain several entries with one operation.

- A$WRITE

  This system call applies only to data files. Your application uses this system call to copy information from a memory buffer and place it in the file. Before using this call, the application can use A$SEEK to position the file pointer at the location within the file to receive the information. The A$WRITE system call requires that the file connection be open. It also requires that the segment of memory from which you copy the data be readable.

- A$TRUNCATE

  This system call can be used only on data files. Your application can use this call to trim information from the end of the file. To do so, the application first must use A$SEEK to position the file pointer at the first byte to be dropped. Then the application invokes the A$TRUNCATE call to drop the specified byte and any bytes located after the specified byte. The A$TRUNCATE system call requires that the file connection be open.

- WAIT$IO

  Your application can use this system call after calling A$READ, A$WRITE, or A$SEEK to receive the concurrent condition code of the prior system call. WAIT$IO can also return the number of bytes read or written.

- A$UPDATE

  This system call forces the Basic I/O System to transfer data remaining in internal buffers immediately to the files on a device. Your application can use this system call to ensure that all files on removable volumes (such as diskettes) are updated before the operator removes the volume.

3.6.5 Obtaining Status

There are two status-related system calls, one for connections and one for files. The calls are A$GET$FILE$STATUS and A$GET$CONNECTION$STATUS. Both of these calls can be used with data files and directory files.

3.6.6 Reading Directory Entries

There are two system calls that your application can use to read entries from a directory. The A$READ system call (which can also be used to read a data file) was discussed earlier, under the heading "Manipulating Data." The second system call is A$GET$DIRECTORY$ENTRY. This system call can be used only on directory files, and can be used without opening a connection.

3.6.7 Deleting and Renaming Files

The Basic I/O System provides one system call for deleting files and another for renaming files. Both of these calls can be used with data files and directory files. The calls are:

- A$DELETE$FILE

  Your application can use this system call to delete data files and directory files. However, any attempt to delete a directory that is not empty will result in an exceptional condition.

  The process of deleting a file involves two stages. First, the application must call A$DELETE$FILE. This causes the file to be marked for deletion. The second stage, which is performed by the Basic I/O System, involves deciding when to delete the file. The Basic I/O System deletes marked files only after all connections to the file have been deleted. Refer to the A$DELETE$CONNECTION system call to see how to delete connections.

- A$RENAME$FILE

  Your application can use this system call to rename both data files and directory files. In renaming a file, your application can move the file to any directory in the same named file tree. For example, you can rename A/B/C to be A/X/C. In effect, this example simply moves File C from Directory B to Directory X. This means that your application can change every component of a file's path name.

3.6.8 Changing Access

The Basic I/O System provides one system call to let your application change a file's access list. This call is A$CHANGE$ACCESS, and it applies to both data files and directories. One rule governs the use of A$CHANGE$ACCESS--only the owner of a file or a user with change entry access to the directory containing the file can change the file's access list.

3.6.9 Identifying a File's Name

The Basic I/O System provides a system call to let your application find out the last component of a file's path name when the application has a connection to the file. The system call is A$GET$PATH$COMPONENT, and you can use it on data files and directories. For an explanation of how you can use this system call repeatedly to obtain the entire path name for a file, see the description of this system call in the *Extended iRMX II Basic I/O System Calls Reference Manual.*

3.6.10 Manipulating Extension Data

When you format a volume to accommodate named files, you have the option of allowing each file to include extension data. The Basic I/O System provides two system calls that allow you to get and set extension data. These calls apply to both data and directory files.

- A$SET$EXTENSION$DATA

  This call provides a means of writing extension data. A$SET$EXTENSION$DATA can be used even if the file connection is not open.

- A$GET$EXTENSION$DATA

  This call provides a means of reading extension data. A$GET$EXTENSION$DATA can be used even if the file connection is not open.

3.6.11 Detecting Changes in Device Status

The Basic I/O System provides the A$SPECIAL system call to allow your application to detect a change in the status of the device containing your named file tree. Specifically, your application can use the "notify" function of the A$SPECIAL system call to establish a mechanism for finding out if the device ceases to be ready. For more information, refer to the A$SPECIAL section of the *Extended iRMX II Basic I/O System Calls Reference Manual.*

3.7 ACCESSING THE GLOBAL TIME-OF-DAY CLOCK

The Basic I/O System provides one system call that obtains the time of day from a battery-powered time-of-day clock (called a global clock), if such a clock is available. Another system call exists to set the global time-of-day clock. The system calls are

- GET$GLOBAL$TIME

  This system call returns the date and time value stored in the global time-of-day system clock.

- SET$GLOBAL$TIME

  This call sets the global date and time values in the global time-of-day clock.

3.8 ACCESSING FILES THROUGH iRMX-NET

The Basic I/O System supports the iRMX NET local area network standard by allowing you to configure the remote file driver and by providing the ENCRYPT system call. This system call encrypts passwords as defined by the iRMX-NET local area network encryption standard.

3.9 CHRONOLOGICAL OVERVIEW OF NAMED FILES

The system calls that can be used with named files cannot be used in arbitrary order. This section provides you with a sense of how the calls relate to one another.

3.9.1 Most Frequently Used System Calls

Figure 3-3 shows the chronological relationships between the most frequently used Basic I/O System calls. To use the figure, start with the leftmost box and follow the arrows. Any path that you can trace is a legitimate sequence of system calls. Keep in mind that this figure does not represent all possible sequences.

3.9.2 Calls Relating to User Objects

The system calls relating to user objects are completely independent of other Basic I/O System calls. With one exception, your application must have a user object before it can use any system call requiring a user object.

Five system calls pertain to user objects. Of the five, GET$DEFAULT$USER and CREATE$USER can be invoked at any time. Two others, DELETE$USER and INSPECT$USER, can be invoked only after user objects exist. The remaining call, SET$DEFAULT$USER requires that both a job and a user object exist.

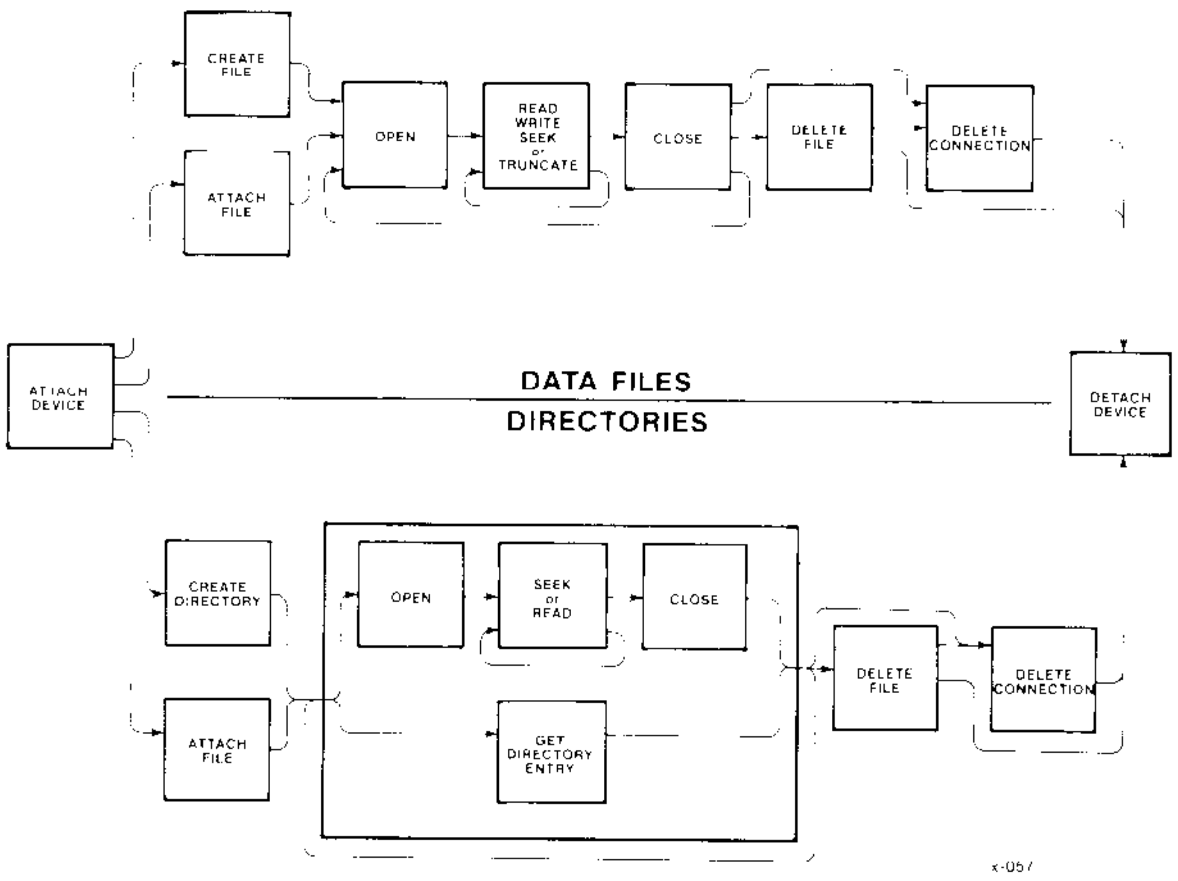


Figure 3-3. Chronology of Frequently Used System Calls for Named Files

3.9.3 Calls Relating to Prefixes

The GET$DEFAULT$PREFIX system call can be invoked whenever a job exists. The SET$DEFAULT$PREFIX, however, requires both a job and a user object.

3.9.4 Calls Relating to Status

Both of the status-related system calls, A$GET$FILE$STATUS and A$GET$CONNECTION$STATUS, can be invoked whenever your application has a file connection.

3.9.5 Calls Relating to Changing Access

The only system call related to changing access, A$CHANGE$ACCESS, can be invoked whenever your application has both a user object and a path or connection to a file.

3.9.6 Calls for Monitoring Device Readiness

There is only one system call that lets your application monitor the readiness of a device, the A$SPECIAL system call. Your application can use the "notify" function of this call any time after your application has obtained a device connection.

3.9.7 Calls Relating to Extension Data

The two system calls relating to extension data, A$GET$EXTENSION$DATA and A$SET$EXTENSION$DATA, can be invoked whenever your application has a connection to a file.

3.9.8 Calls for Renaming Files

The one call for renaming a file, A$RENAME$FILE, can be used whenever your application has a connection to the file to be renamed, a user object, and a path that is to become the new pathname.

3.9.9 Calls for Identifying File Names

There is only one system call for finding out a file's name, A$GET$PATH$COMPONENT. Your application can use this call whenever the application has a connection to the file.

3.9.10 Calls for iRMX-NET

The BIOS system call ENCRYPT encrypts a password to enable remote file access through iRMX NET. You must use this call to encrypt a password. The ENCRYPT system call can also be used by any application that needs to perform password encryption. Password decryption is not supported.

intel®

CHAPTER 4 PHYSICAL FILES

4.1 INTRODUCTION

The Basic I/O System provides physical files to allow your applications to read (or write) strings of bytes from (or to) a device. A physical file occupies an entire device, and the Basic I/O System provides your applications with the ability to capitalize on the physical characteristics of the device.

4.2 SITUATIONS REQUIRING PHYSICAL FILES

The close relationship between a device and a physical file is particularly useful when your application uses sequential devices. For example, you should use physical files to communicate with line printers, display tubes, plotters, magnetic tape units, and robots.

There are even some instances where you should use physical files to communicate with random devices such as disks, diskettes, and bubble memories. For instance

- Formatting Volumes

  Whenever you create an application to format a disk or diskette, the application must have access to every byte on the volume. Only physical files provide this kind of access.

- Volumes in Formats Required by Other Systems

  If your application must read or write volumes that have been formatted for systems other than the Basic I/O System, you must use physical files. Your application will have to interpret such information as labels and file structures. A physical file can provide your application with access to the raw information.

- Implementing Your Own File Format

  Suppose that your application requires a less sophisticated file structure than that provided by iRMX II named files. You can build a custom file structure using a physical file as a foundation.

4.3 CONNECTIONS AND PHYSICAL FILES

Although there is a one-to-one correspondence between the bytes on a device and the bytes of a physical file, the device connection is different than the file connection. The Basic I/O System maintains this distinction to remain consistent with named files and stream files. This consistency helps you develop applications that can use any kind of file.

4.4 USING PHYSICAL FILES

Several system calls can be used with physical files, but the order in which they are used is not arbitrary. The following list provides a brief description (in chronological order) of what an application must do to use a physical file.

1. Obtain a device connection.

   Your application must call A$PHYSICAL$ATTACH$DEVICE to obtain a device connection for the device. This needs to be done only once for each device and is necessary for two reasons. When your application creates the physical file, the device connection tells the Basic I/O System which device is to contain the file and also that the file must be a physical file.

2. Obtain a file connection.

   If your application knows that the file has not yet been created, it should use the A$CREATE$FILE system call to obtain a file connection. This will work even if the physical file has already been created. Use the token of the device connection as the PREFIX parameter to tell the Basic I/O System which device you want as your physical file.

   If, on the other hand, your application is certain that the file has already been created, use the A$ATTACH$FILE system call to obtain the file connection. To do this, your application can use either the device connection for the device or an existing file connection to the file as the PREFIX parameter in the system call.

   This careful distinction between the A$CREATE$FILE and the A$ATTACH$FILE system calls is necessary to be consistent with named files. If you want your application to work with any kind of file, you must maintain this consistency.

3. Open the file connection.

   Use the A$OPEN system call to open the connection. When opening the connection, your application must specify how the file can be shared and how the application uses the connection.

4. Manipulate the file.

   Four system calls can be used to read, write, or otherwise manipulate your physical file:

   - The A$READ and A$WRITE system calls can be used to read from the device and write to the device, respectively.
   - The A$SEEK system call can be used to manipulate the file connection's file pointer if the device is a random device such as disk, diskette, or bubble.
   - The A$SPECIAL system call can be used to request device-dependent functions from the device driver. The precise nature of these functions depends upon the kind of device and the number of special functions supported by the device driver. Be aware that use of special functions can prevent an application from being device-independent.

5. Close the file connection.

   Use the A$CLOSE system call to close the connection. This is particularly important if the share mode of the connection restricts the use of the file through other connections. Note that your application can repeat steps 2, 3, and 4 any number of times.

6. Delete the file connection.

   Use the A$DELETE$CONNECTION system call to delete the file connection. This is only necessary if your application is completely finished using the file.

7. Request that the device be detached.

   Invoke the A$PHYSICAL$DETACH$DEVICE system call to let the Operating System know when your application is finished using the device. The Operating System keeps track of the number of applications using the device and avoids detaching it until it is no longer being used by any application. Only then does the Operating System actually detach the device.

All of these system calls are described in the *Extended iRMX II Basic I/O System Calls Reference Manual.*

CHAPTER 5
STREAM FILES

5.1 INTRODUCTION

Stream files provide a means for one task to send large amounts of information to another task. Be aware that this is one of several techniques for job-to-job communication. If you are not familiar with other techniques, refer to the *Extended iRMX II Programming Techniques Manual.*

The aspect of stream files that makes them very useful is that they allow a task to communicate with a second task as though the second task were a device. This extends the notion of device independence to include tasks.

Because two tasks are involved in using each stream file, each task must perform one half of a protocol. There are several protocols that work, but the following one is typical and serves as a good illustration. Note that the two halves of the protocol can be performed in either order or concurrently.

5.2 ACTIONS REQUIRED OF THE WRITING TASK

The writing task must perform seven steps in its half of the protocol to ensure that it has established communication with the reading task. The steps are

1. Obtain a connection to the stream file device.

   Although stream files do not actually require a physical device, your application must call A$PHYSICAL$ATTACH$DEVICE to obtain a device connection before creating a stream file. This is necessary because, when your application invokes the A$CREATE$FILE system call, the device connection tells the Basic I/O System what kind of file to create.

   The A$PHYSICAL$ATTACH$DEVICE system call requires a parameter that identifies the device to be attached. For stream files, there is only one device, and its name is specified during the process of configuring the system. Intel recommends the name "STREAM", but it is possible that the person responsible for configuring your system changed this name. For the remainder of this discussion, this manual assumes that the name of your system's stream file device is "STREAM".

As with other devices, "STREAM" cannot be multiply attached, so the system program should be written so as to call A$PHYSICAL$ATTACH$DEVICE only once. The program can then save the device connection and pass it to any application program that requests it.

2. Create the stream file.

   Use the A$CREATE$FILE system call with the device connection to create the stream file and obtain a token for a file connection to the stream file. Use the token for the device connection as the PREFIX parameter, in order to tell the Basic I/O System to create a stream file.

3. Pass the file connection to the reading task.

   There are several ways of doing this, including the use of object directories and mailboxes. For explicit instructions, refer to the *Extended iRMX II Programming Techniques Manual.*

4. Open the file for writing.

   Use the A$OPEN system call to open the file connection for writing. Set the CONNECTION parameter equal to the token for the file connection; set the MODE parameter for writing; and set the SHARE parameter for sharing only with readers.

5. Write information to the stream file.

   Use the A$WRITE system call as often as needed to write information to the stream file. Use the token for the file connection as the CONNECTION parameter.

   The Basic I/O System uses the concurrent part of the A$WRITE system call to synchronize the writing and reading tasks on a call-by-call basis. The Basic I/O System does this by sending a response to each invocation of A$WRITE only after the reading task has finished reading all information that was written by the A$WRITE call.

6. Close the connection.

   When finished writing to the stream file, use the A$CLOSE system call to close the connection. Note that after this step, the writing task can repeat steps 4, 5, and 6 as many times as needed.

7. Delete the connection.

   Use the A$DELETE$CONNECTION system call to delete the connection to the stream file.

All of these system calls are described in the *Extended iRMX II Basic I/O System Calls Reference Manual.*

5.3 ACTIONS REQUIRED OF THE READING TASK

The reading task must perform the following six steps in its half of the protocol to successfully read the information written by the writing task.

1. Get the file connection for the stream file.5-;

   The technique used to accomplish this depends on how the writing task passed the file connection.

2. Create a second file connection for the stream file.

   There are two reasons for doing this. First, the reading task must have a different file pointer than that of the writing task. Second, the Basic I/O System rejects any connections created in one job but used by another to manipulate a file.

   Obtain this new connection by using the A$ATTACH$FILE system call. Set the PREFIX parameter to the token for the original file connection.

   NOTE

   The reading task can also use the A$CREATE$FILE system call to obtain the new connection to the same stream file. The reason for this is that the Basic I/O System examines the nature of the PREFIX parameter in the A$CREATE$FILE system call. If the value provided is a device connection, the Basic I/O System will create a new file and return a connection for it. On the other hand, if the value provided is a file connection, the Basic I/O System will just create another connection to the same file.

   This careful distinction between the A$CREATE$FILE and the A$ATTACH$FILE system calls is necessary to be consistent with named and physical files. If you want your application to work with any kind of file, you must maintain this consistency.

3. Open the new file connection for reading.

   Use the A$OPEN system call to open the connection for reading. Set the CONNECTION parameter equal to the token for the new connection. Set the MODE parameter for reading, and set the SHARE parameter for sharing with all connections to the file.

4. Commence reading.

   Use the A$READ system call to read the file until reading is no longer necessary or until an end-of-file condition is detected by the Basic I/O System.

5. Close the new file connection.

   Use the A$CLOSE system call to close the new file connection. Note that after this step, the reading task can repeat steps 3, 4, and 5 as many times as needed.

6. Delete the new file connection.

   Use the A$DELETE$CONNECTION system call to delete the new connection to the stream file. The writing task deletes the old connection, and, as soon as both connections have been deleted, the Basic I/O System deletes the stream file.

All of these system calls are described in the *Extended iRMX II Basic I/O System Calls Reference Manual.*

CHAPTER 6 SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

6.1 INTRODUCTION

This chapter provides you with background information on the system calls of the Basic I/O System. For detailed information on system call parameters, refer to the *Extended iRMX II Basic I/O System Calls Reference Manual.*

BIOS system calls can be divided into two categories according to their names. The first category consists of system calls having names of the form

RQ$XXXXX

where XXXXX is a brief description of what the system call does. The second category consists of system calls having names of the form

RQ$A$XXXXX

System calls of the first category, without the A, are synchronous calls. They begin running as soon as your application invokes them, and continue running until they detect an error or accomplish everything they must do. Then they return control to your application. In other words, synchronous calls act like subroutines.

System calls of the second category (those with the A) are called asynchronous because they accomplish their objectives by using tasks that run concurrently with your application. This allows your application to accomplish some work while the Basic I/O System deals with devices such as disk drives and tape drives.

6.2 SYNCHRONOUS SYSTEM CALLS

The following paragraphs explain properties of certain input parameters to synchronous Basic I/O System calls.

6.2.1 User Parameter

This parameter is specified in many synchronous system calls (and in some asynchronous ones as well). It contains a token designating the caller's user object. A SELECTOR$OF(NIL) specification designates the default user. The Basic I/O System ignores this parameter for physical and stream files.

6.2.2 File-Path Parameter(s) for Named Files

Named files are designated in system calls by specifying their path, that is, their prefix and subpath. The prefix parameter can be a token designating an existing device connection or file connection. If this parameter is SELECTOR$OF(NIL), the default prefix for the calling task's job is assumed.

For named files, the subpath parameter is a pointer to an ASCII string. The form of this string is described in the following paragraph. The subpath can also be NIL or can point to a null string, in which case a prefix indicates the desired connection. For physical and stream files, the subpath parameter is always ignored.

NOTE

A file connection that was obtained in one job cannot be used as a connection by another job. However, a file connection can be used as a prefix by other jobs in any call requiring prefix and subpath parameters. (The only exceptions to this rule are that the other jobs cannot use the connection as a prefix while specifying a null subpath in calls to A$CHANGE$ACCESS or A$DELETE$FILE.) This means that a file connection can be passed to another job and the other job can obtain its own connection to the same file by calling A$ATTACH$FILE, with the passed file connection being used as the prefix parameter in the call.

System calls referring to named files can specify paths in the following forms:

Prefix	Subpath	Designated Connection
0	a pointer to a null string	Connection whose token is the default prefix.
0	Pointer to ASCII string	ASCII string defines a path from the connection whose token is the default prefix to the target connection.
token	a pointer to a null string	Connection whose token is contained in the prefix.
token	Pointer to ASCII string	Prefix parameter contains a token for a connection. ASCII string defines a path from that connection to the target connection.

The subpath ASCII string is a list of file names separated by slashes, terminating with the desired file. A file name can be 1-14 ASCII characters, including any printable ASCII character except the slash (/) and up-arrow () or circumflex (^). In Figure 6-1, for example, if the prefix is the token for directory OBSTETRICS and we wish to reference file OUT-PATIENT, the subpath parameter must point to the string

DELIVERY/POST-PARTUM/OUT-PATIENT

If the ASCII string begins with a slash, the prefix merely designates the tree and the subpath is assumed to start at the root directory of the tree associated with the prefix. For example, if the prefix designates directory GYNECOLOGY in Figure 6-1, the subpath to OUT-PATIENT is

/OBSTETRICS/DELIVERY/POST-PARTUM/OUT-PATIENT

Named files can also be addressed relative to other files in the tree, using "↑" or "/" as a path component. (These two symbols have the same meaning. Some terminals do not have the up-arrow key.) The "↑" or "/" refers to the parent directory of the current file in the path scan. For example, now that we have a connection to OUT-PATIENT in Figure 6-1, we can use that connection to specify a subpath to IN-PATIENT. With the token for the OUT-PATIENT connection as our prefix, the subpath string would be

/IN-PATIENT

Note that no slash follows the "/" in this example.

Of course an even simpler approach would be to designate directory POST-PARTUM as the prefix, in which case the ASCII string becomes

    IN-PATIENT

NOTE

The Basic I/O System does not distinguish between uppercase and lowercase characters in subpaths. For example, the subpath "xyz" is equivalent to the subpath "XYZ".

6.2.3 Response Mailbox Parameter

This parameter is specified only in asynchronous system calls. It contains a token designating the mailbox that is to receive the result of the call. This information is provided by tasks to synchronize parallel operations. To receive the result of the call, a task must either call RECEIVE$MESSAGE and wait at the designated mailbox or call WAIT$IO. Be aware that if several calls share the same mailbox, the results may be received out of order.

Most asynchronous system calls return only an I/O result segment to the response mailbox. This segment contains an exception code and other information about the operation. Appendix C describes the I/O result segment. Other system calls--A$ATTACH$FILE, A$CREATE$DIRECTORY, A$CREATE$FILE, and A$PHYSICAL$ATTACH$DEVICE--return to the mailbox a token for a connection if the system call performs successfully or an I/O result segment otherwise. After calling RECEIVE$MESSAGE to obtain the result of one of these system calls, a task should perform a GET$TYPE system call to ascertain the type of object returned to the response mailbox. The *Extended iRMX II Nucleus User's Guide* describes GET$TYPE in detail.

NOTE

I/O result segments should be deleted when they are no longer needed. They remain in memory until deleted.

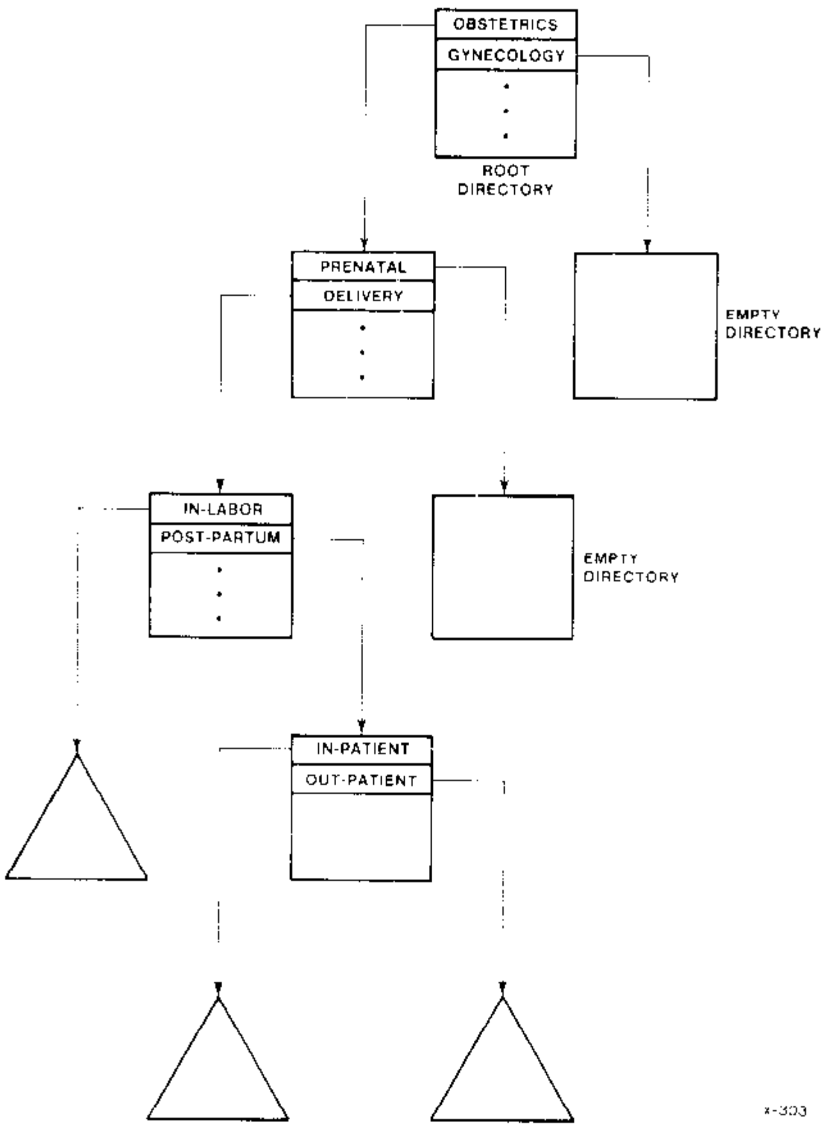


Figure 6-1. Sample Named File Tree

6.2.4 I/O Buffers

The A$READ and A$WRITE system calls each require a buffer to read from or write to while performing I/O. When you create these buffers, bear in mind the following restrictions:

- The memory segments used for the I/O buffers must have the appropriate access rights. For example, if you are going to read data from an I/O device and place it into a buffer, the memory segment must have write access. Likewise, if you are going to take data from a buffer and write it to an I/O device, the memory segment must have read access.
- Once the I/O operation has been invoked, the tasks of your application should avoid changing the contents of the buffer until the Basic I/O System finishes the operation.
- If you use an iRMX II segment as a buffer, be sure that the buffer is not deleted while an I/O operation is in progress.
- If you choose to use an iRMX II segment as a buffer, you must ensure that the segment is in the same job as the task performing the I/O operation. Using segments from one job as buffers for I/O operations in a different job can lead to a problem. For instance, suppose that Job A owns an iRMX II segment, and that Job B uses this segment as a buffer for I/O. If Job A is deleted, the iRMX II Operating System automatically deletes the buffer even if Job B has I/O in progress.

  Note that the problem of unintentional deletion of objects shared between jobs exists for all objects when the job that owns the object is deleted.

6.3 ASYNCHRONOUS SYSTEM CALLS

Each asynchronous system call has two parts--one sequential, and one concurrent. As you read the descriptions of the two parts, refer to Figure 6-2 to see how the parts relate.

- the sequential part

  The sequential part behaves in much the same way that fully synchronous system calls do. Its purpose is to verify parameters, check conditions, and prepare the concurrent part of the system call. The sequential part then returns control to your application.
- the concurrent part

  The concurrent part runs as an iRMX II task. The task is made ready by the sequential part of the call, and it runs only when the priority-based scheduling of the iRMX II Operating System gives it the processor.

The reason for splitting the asynchronous calls into two parts is performance. The functions performed by these calls are somewhat time-consuming because they usually involve mechanical devices. By performing these functions concurrently with other work, the Basic I/O System allows your application to run while the Basic I/O System waits for the mechanical devices to respond to your application's request.

Let's look at a brief example showing how your application can use asynchronous calls. Suppose your application requires some information that is stored on disk. The application issues the A$READ system call to have the Basic I/O System read the information into memory. Let's trace the action one step at a time:

1. Your application issues the A$READ system call. This call requires, as do all asynchronous calls, that your application specify a response mailbox for communication with the concurrent part of the system call.
2. The sequential part of the A$READ call begins to run. This part checks the parameters for validity.
3. If the sequential part of the call detects a problem, it signals an exception and returns control to your application. It does not make ready the Basic I/O System task to perform the reading function.
4. Your application receives control. Its actions at this point depend on the condition code returned by the sequential part of the system call. Therefore, the application tests the condition code. If the code is E$OK, the application continues running until it must have the information from the disk. It is at this point that your application can take advantage of the asynchronous and concurrent behavior of the Basic I/O System.

   For example, your application can implement double (or multiple) buffering by issuing another (or several) A$READ system call(s) while waiting for the first call to finish running. Alternatively, your application can use this overlapping processing to perform computations. The point is that you can decide what you want your application to do while the asynchronous system call is running.

   On the other hand, if your application finds that the condition code returned from the sequential part of the system call is other than E$OK, the application can assume that the Basic I/O System did not make ready a task to perform the function.

   For the balance of this example, we will assume that the sequential part of the system call returned an E$OK condition code.

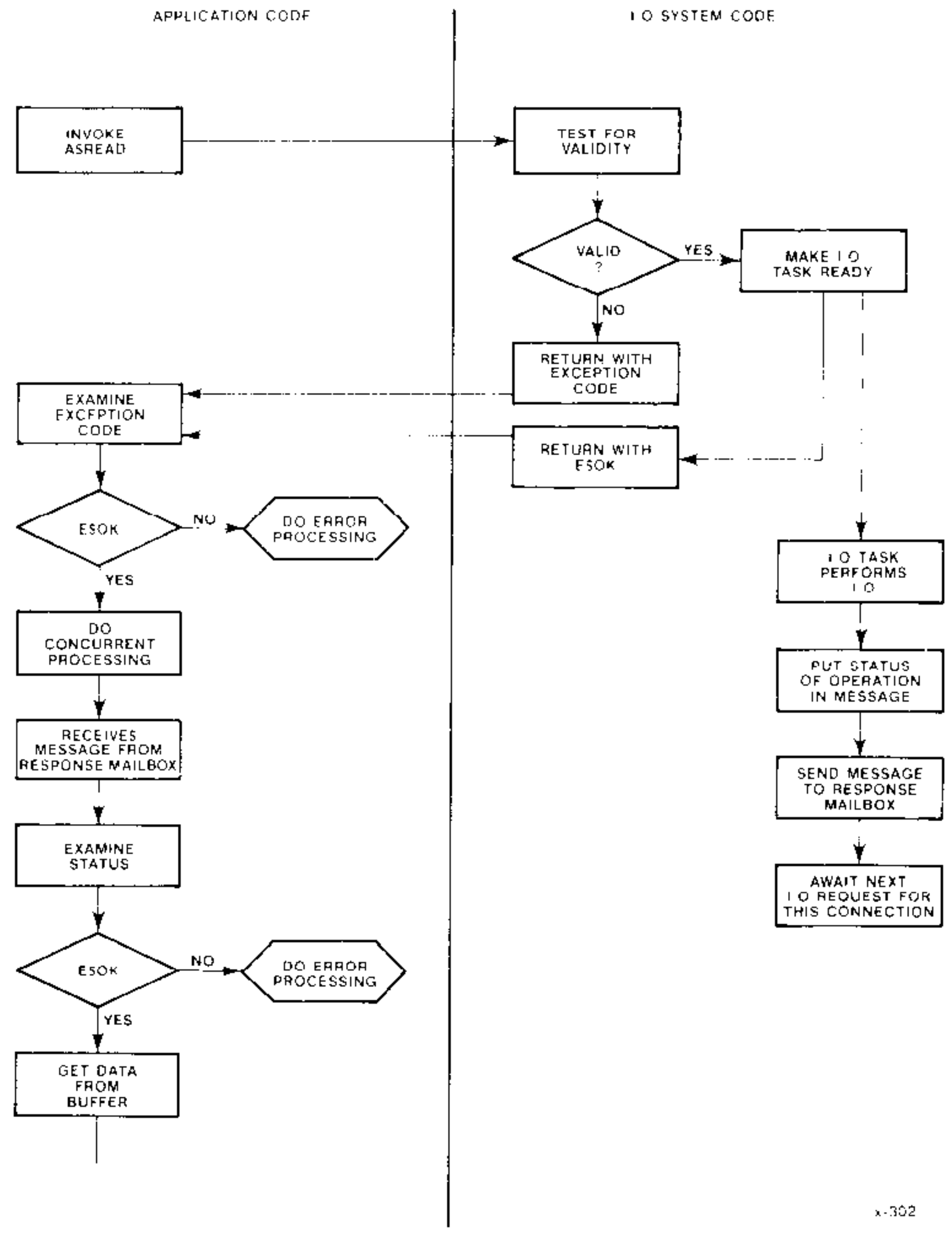


Figure 6-2. Concurrent Behavior of an Asynchronous System Call

5. Your application now must have the information. Before taking the information from the buffer, your application must verify that the concurrent part of the A$READ system call ran successfully. There are two ways in which the task can do this. One way is for the application to issue a RECEIVE$MESSAGE system call to check the response mailbox that the application specified when it invoked the A$READ system call. The other way (which can be used only after a call to A$READ, A$WRITE, or A$SEEK) is for the application to issue a WAIT$IO system call, in which it passes a token for the response mailbox and receives the concurrent condition code directly.

   By using the RECEIVE$MESSAGE system call, the application obtains a segment that contains, among other things, a condition code for the concurrent part of the A$READ system call. If this condition code is E$OK, then the reading operation was successful, and the application can get the data from the buffer. On the other hand, if the code is not E$OK, the application should analyze the code and attempt to ascertain why the reading operation was not successful.

   By using the WAIT$IO system call, the application receives directly the condition code for the concurrent part of the A$READ system call. The application also receives directly another value. If the concurrent condition code for A$READ is E$OK, then this other value is the number of bytes successfully read; otherwise, this other value has no significance.

In the foregoing example, we used a specific system call (A$READ) to show how asynchronous calls allow your application to run concurrently with I/O operations. Now let's look at some generalities about asynchronous calls.

- All asynchronous system calls consist of two parts--one sequential and one concurrent. The Basic I/O System will activate the concurrent part only if the sequential part runs successfully (returns E$OK).
- Every asynchronous system call allows your application to designate a response mailbox through which the application receives the result of the concurrent part of the system call.
- Whenever the sequential part of an asynchronous system call returns a condition code other than E$OK, your application should not attempt to receive a message from the response mailbox, nor should it call WAIT$IO. There can be no further information for the application because the Basic I/O System cannot run the concurrent part of the system call.
- Whenever the sequential part of an asynchronous system call runs successfully (E$OK), your application can count on the Basic I/O System running the concurrent part of the system call. Your application can take advantage of the concurrency by doing some processing before receiving the message from the response mailbox or before calling WAIT$IO.

- Whenever the concurrent part of a system call runs, the Basic I/O System signals its completion by sending an object to the response mailbox. The precise nature of the object depends upon which system call your application invoked. You can find out what kind of object comes back from a particular system call by looking up the call in the *Extended iRMX II Basic I/O System Calls Reference Manual.* If more than one type of object can be returned, your application can ascertain the type of the returned object by calling GET$TYPE.
- Whenever the Basic I/O System returns a segment to your application's response mailbox and the application calls RECEIVE$MESSAGE to obtain information from that segment, the application should delete the segment when the segment is no longer needed. The Basic I/O System draws memory for such segments from the memory pool of the calling task's job, so if the application fails to delete such segments, large amounts of memory are wasted on unneeded segments.
- If your application calls WAIT$IO to obtain the results of a call to A$READ, A$WRITE, or A$SEEK, the application does not have access to the I/O result segment and therefore cannot delete it. While this seems to be a problem at first glance, it is actually an advantage. It enables the Basic I/O System to maintain a supply of I/O result segments that it can use repeatedly, instead of creating a separate I/O result segment for each A$READ, A$WRITE, or A$SEEK. Because most I/O-related operations are reads, writes, or seeks, this means a significant performance enhancement for your application.

6.4 CONDITION CODES

The Basic I/O System returns a condition code when a system call is invoked. If the call executes without error, the Basic I/O System returns the code "E$OK." If an error is encountered, some other code is returned.

For those system calls that do not require a response mailbox parameter, the Basic I/O System returns the condition code to the word pointed to by the except$ptr parameter. If an exceptional condition occurs, the Basic I/O System can then either return control to the calling task or pass control to an exception handler. See the *Extended iRMX II Nucleus User's Guide* for a detailed description of exception handling.

For those system calls that do require a response mailbox parameter (the asynchronous calls), the Basic I/O System returns a condition code for the sequential portion of the call to the word pointed to by the except$ptr parameter and a condition code for the concurrent portion of the call to the status field of the I/O result segment (see Appendix C). If a sequential exceptional condition occurs, the Basic I/O System either returns control to the calling task or passes control to an exception handler. It does not process the asynchronous portion of the call. If a concurrent exceptional condition occurs, the calling task must signal the exception handler or process the exceptional condition in line.

CHAPTER 7 CONFIGURING THE BASIC I/O SYSTEM

7.1 INTRODUCTION

The Basic I/O System is a configurable layer of the iRMX II Operating System. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, Intel provides three kinds of information:

- A list of configurable options
- Detailed information about the options
- Procedures to allow you to specify your choices

The balance of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

7.2 BASIC I/O SYSTEM CALLS

You can select the timers, clocks, and drivers required by your application. The advantage in being able to do this is that you can reduce the amount of Basic I/O System code needed to support your application. With the Interactive Configuration Utility (ICU), you can exclude entire file drivers, such as the stream file driver.

7.3 INTEL I/O DEVICES

You must specify which Intel I/O devices (controllers) are part of your hardware configuration. The devices that you can specify are listed in the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

For each device that you select, you must specify a name, physical characteristics, and desired operating modes.

7.4 BUFFERS

For each device, you must specify the number of buffers that the Basic I/O System is to manage during I/O operations on that device.

7.5 TIMING FACILITIES

You must specify whether you want your system to include the timing facilities related to the SET$TIME, SET$GLOBAL$TIME, GET$TIME, and GET$GLOBAL$TIME system calls.

7.6 SERVICE TASK PRIORITIES

You must specify the priorities of the Basic I/O System tasks that attach devices and delete connections.

7.7 CREATING A FILE WITH AN EXISTING PATHNAME

Occasionally, a task will call A$CREATE$FILE, specifying a pathname that is identical to the pathname of a file that already exists. The Basic I/O System provides a configuration parameter (called NO CREATE FILE) that enables you to specify what should happen in this case.

If NO CREATE FILE is selected, a call to A$CREATE$FILE will return the exception code E$FEXIST, regardless of the value of the must$create parameter in the call.

If NO CREATE FILE is not selected, then what happens depends upon the value of the must$create parameter in the call to A$CREATE$FILE. If must$create is true (0FFH), then the Basic I/O System returns the E$FEXIST exception code. If must$create is false (0), then the existing file is truncated or expanded, according to the size parameter in the call to A$CREATE$FILE.

7.8 SYSTEM MANAGER ID

You must specify whether you want a system manager (user).

7.9 SYSTEM INITIALIZATION ERROR REPORTING

During the configuration process, you can elect to have the system report BIOS initialization errors. If you configure System Initialization Error Reporting (SIER) into your application system when you configure the Nucleus, the Operating System reports initialization errors from all layers of the Operating System. On encountering a BIOS initialization error, it gives control to the monitor after writing the following message to the monitor console:

```
BIOS Initialization Error: <error code number>
```

If System Initialization Error Reporting is not configured into the system, the original BIOS initialization task places the BIOS ID code (2) and the corresponding error code into the first two words of the Nucleus data segment (1E0:0000H). If no monitor is configured, it then goes into an infinite error loop.

7.10 FACTORS AFFECTING BASIC I/O SYSTEM PERFORMANCE

The purpose of this section is to make you aware of the factors that have the greatest impact on the performance (speed) of the Basic I/O System. Note that you determine some of these factors during software configuration, but you determine other factors at other times. The factors are as follows:

- Device granularity, which is the smallest number of bytes that can be read from or written to a device in a single I/O operation. If this value is selectable, you determine it either by jumpering hardware or by means of software, depending upon the device.
- Volume granularity, which is the smallest number of contiguous bytes that can be allocated from a volume in a single allocation. This value can vary from volume to volume and must be a multiple of the device granularity. You specify it when formatting the volume with the FORMAT command of the Human Interface.
- File granularity, which is the smallest number of bytes that can be allocated to a file in a single allocation. This value can vary from file to file and must be a multiple of the volume granularity. You specify each file's granularity when creating the file with the A$CREATE$FILE system call.
- The number of buffers for each device-unit. You specify this value when configuring the Basic I/O System.
- The number of bytes to be read or written. You specify this value in calls to A$READ and A$WRITE.
- The amount of time between updates performed by the fixed update and timeout update features. You specify these time intervals when configuring the Basic I/O System. These two kinds of updating are explained in the *Extended iRMX II Basic I/O System Calls Reference Manual* in the description of the A$UPDATE system call.

For best results with these factors, you should begin by using your best judgment. Then, using the resulting performance figures as a base, you can experiment by changing a few (perhaps only one) factors at a time.

Obtaining the optimum combination of these factors is vital to the performance of any application of which I/O operations are a major part. Testing system performance with various combinations can result in a system with higher performance.

APPENDIX A iRMX® II DATA TYPES

A.1 DATA TYPES

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

BOOLEAN	A BYTE that is considered to have a value of TRUE if it is 0FFH, and FALSE if it is 00H. In PL/M-286, DECLARE BOOLEAN LITERALLY 'BYTE';
BYTE	An unsigned eight-bit binary number.
DWORD	An unsigned four-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 an 80286 segment.
POINTER	Two consecutive WORDs containing the selector of an 80286 segment and an offset into that segment. The offset must be in the word having the lower address.
SELECTOR	An index into a descriptor table that identifies a particular memory segment. The descriptor table entry lists the segment's base, limit, type, and privilege level.
STRING	A sequence of consecutive BYTEs. The value contained in the first byte is the number of bytes that follow it in the string.
TOKEN	A SELECTOR that contains the logical address of an object. The selector refers to an entry in the descriptor table that lists the physical address of the object. A token must be declared literally a SELECTOR.
WORD	An unsigned two-byte binary number.

APPENDIX B OBJECT TYPES AND RESOURCE REQUIREMENTS

B.1 INTRODUCTION

This appendix lists the type codes for all iRMX II objects. In addition, it documents the amount of memory needed to create Basic I/O System objects.

B.2 OBJECT TYPES

Each iRMX II object type is known within iRMX II systems by means of a numeric code. Table B-1 lists the types with their codes.

Table B-1. Type Codes

OBJECT TYPE	NUMERIC CODE
Job	1
Task	2
Mailbox	3
Semaphore	4
Region	5
Segment	6
Extension	7
Composite	8
User	100
Connection	101
I/O Job	300
Logical Device	301
User-Created Composite	varies from 8000H to 0FFFFH depending on the value specified in CREATE$EXTENSION

The first eight objects, plus user-created composites, are described in the *Extended iRMX II Nucleus User's Guide*. User and connection object types are described in Chapter 4 of this manual. I/O jobs and logical devices are described in the *Extended iRMX II Extended I/O System User's Guide*.

B.3 RESOURCE REQUIREMENTS

The Basic I/O System obtains memory from the calling job's memory pool when creating objects. The values listed here reflect Release 3 of the iRMX II Operating System.

Object	Number of 16-byte paragraphs required by the Basic I/O System
I/O Result Segment	4 (5 for an internal IORS that the Operating System creates when attaching a device)
Connection (to named file)	6
Connection (to physical file)	4
User object	3 (minimum)

APPENDIX C I/O RESULT SEGMENT

C.1 OVERVIEW

Certain asynchronous I/O system calls return a data structure called an I/O Result Segment to the mailbox specified by the "resp$mbox" parameter. The following system calls can return such a segment:

A$ATTACH$FILE
A$CHANGE$ACCESS
A$CLOSE
A$CREATE$DIRECTORY
A$CREATE$FILE
A$DELETE$CONNECTION
A$DELETE$FILE
A$OPEN
A$PHYSICAL$ATTACH$DEVICE
A$PHYSICAL$DETACH$DEVICE
A$READ
A$RENAME$FILE
A$SEEK
A$SPECIAL
A$TRUNCATE
A$UPDATE
A$WRITE

Four of these system calls (A$ATTACH$FILE, A$CREATE$DIRECTORY, A$CREATE$FILE, and A$PHYSICAL$ATTACH$DEVICE) can return either a connection or an I/O result segment to the mailbox. Your application task can determine which type of object has been returned by making a GET$TYPE system call before trying to examine the object.

Before waiting at the response mailbox to receive the I/O result segment, your application task should examine the condition code returned in the word pointed to by the "except$ptr" parameter. If this code is "E$OK", the task can wait at the mailbox. However, if the code is not "E$OK", an exceptional condition exists and nothing is sent to the mailbox.

Immediately after receiving the I/O result segment, the task should examine the status field. This field contains an "E$OK" if the system call was completed successfully or an exceptional-condition code if an error occurred. The result segment also contains the actual number of bytes read or written, if appropriate.

C.2 STRUCTURE OF I/O RESULT SEGMENT

The I/O result segment is structured as follows:

```
DECLARE   iors          STRUCTURE(
    status              WORD,
    unit$status         WORD,
    actual              WORD);
```

where

status	Condition code indicating the outcome of the call. Appendix D lists these asynchronous condition codes.
unit$status	The lower four bits of this field contain device-dependent error code information that is meaningful only if the status is E$IO. The codes, their meanings, and their associated mnemonics are as follows:

Code	Mnemonic	Meaning
0	IO$UNCLASS	An error occurred for which it was impossible to ascertain the cause.
1	IO$SOFT	Soft error; the I/O system has retried the operation and failed; another retry is not possible.
2	IO$HARD	Hard error; a retry is not possible.
3	IO$OPRINT	Operator intervention is required.
4	IO$WRPROT	Write-protected volume.
5	IO$NO$DATA	No data on the next tape record.

Code	Mnemonic	Meaning
6	IO$MODE	A read (or write) was attempted before the previous write (or read) completed.
7	IO$NO$SPARES	An I/O error occurred during disk formatting; no alternate tracks were available.
8	IO$ALT$AS-SIGNED	An I/O error occurred during disk formatting; an alternate track was assigned.

actual The actual number of bytes transferred.

The I/O result segment contains other fields which are of interest only to the designer of a device driver. Refer to the *Extended iRMX II Device Drivers User's Guide* for information about the remaining fields in the I/O result segment.

C.3 UNIT STATUS FOR SPECIFIC DEVICES

You may need to know the information contained in the "unit$status" field for the following devices.

C.3.1 iSBC® 214/215G Controller

Under certain circumstances, the iSBC 215 Winchester disk controller and the iSBC 214 diskette, disk, and tape controller place information in the high twelve bits of this word. If the low four bits indicate IO$SOFT, the controller sets the high twelve bits as follows:

Bit	Interpretation
15 (leftmost)	1 = seek error
14	1 = cylinder address miscompare
13	1 = drive fault
12	1 = ID field ECC error
11	1 = data field ECC error
10-8	unused
7	1 = sector not found
6-4	unused

On the other hand, if the low four bits indicate IO$HARD, the iSBC 215G and iSBC 214 controllers set the high 12 bits as follows:

Bit	Interpretation
15	1 = invalid address
14	1 = sector not found
13	1 = invalid command
12	1 = no index
11	1 = diagnostic fault
10	1 = illegal sector size
9	1 = end of media
8	1 = illegal format type
7	1 = seek in progress
6	1 = ROM error
5	1 = RAM error
4	unused

If you need more detailed information regarding the meaning of these errors, refer to the *iSBC®15 Winchester Disk Controller Hardware Reference Manual* or to the *iSBC®214 Multi-Peripheral Controller Hardware Reference Manual.*

APPENDIX D EXCEPTION CODES

D.1 OVERVIEW

This appendix lists two types of exception codes. Those detected synchronously with system call invocation (sequential codes) and those detected during the asynchronous portion of system call processing (concurrent codes). Exception conditions are further classified into programmer errors and environmental conditions. A programmer error is a condition that is preventable by the calling task. An environmental condition is an exception condition caused by circumstances beyond the control of the calling task. The sequential codes are returned to the location addressed by the "except$ptr" field of the system call. The concurrent codes are returned in an I/O result segment (see Appendix C). This appendix lists all codes with their decimal and hexadecimal equivalents.

D.2 SEQUENTIAL (ENVIRONMENTAL) EXCEPTION CODES

CODE	DECIMAL	HEXADECIMAL
E$OK	0	0H
E$TIME	1	1H
E$MEM	2	2H
E$LIMIT	4	4H
E$EXIST	6	6H
E$NOT$CONFIGURED	8	8H
E$SUPPORT	35	23H
E$DEV$OFF$LINE	46	2EH
E$IFDR	47	2FH
E$NOT$FILE$CONN	50	32H
E$NOT$DEVICE$CONN	51	33H
E$BUFFERED$CONN	54	36H
E$NOT$SAME$DEVICE	58	3AH
E$PATHNAME$SYNTAX	62	3EH

D.3 SEQUENTIAL (PROGRAMMER ERROR) EXCEPTION CODES

CODE	DECIMAL	HEXADECIMAL
E$TYPE	32770	8002H
E$PARAM	32772	8004H
E$NOUSER	32801	8021H
E$NOPREFIX	32802	8022H
E$BAD$BUFF	32803	8023H

D.4 CONCURRENT (ENVIRONMENTAL) EXCEPTION CODES

CODE	DECIMAL	HEXADECIMAL
E$OK	0	0H
E$MEM	2	2H
E$FEXIST	32	20H
E$FNEXIST	33	21H
E$DEVFD	34	22H
E$SUPPORT	35	23H
E$EMPTY$ENTRY	36	24H
E$DIR$END	37	25H
E$FACCESS	38	26H
E$FTYPE	39	27H
E$SHARE	40	28H
E$SPACE	41	29H
E$IDDR	42	2AH
E$IO	43	2BH
E$FLUSHING	44	2CH
E$ILLVOL	45	2DH
E$DEV$OFFLINE	46	2EH
E$IFDR	47	2FH
E$FRAGMENTATION	48	30H
E$DIR$NOT$EMPTY	49	31H
E$NOT$FILE$CONN	50	32H
E$NOT$DEVICE$CONN	51	33H
E$CONN$NOT$OPEN	52	34H
E$CONN$OPEN	53	35H
E$BUFFERED$CONN	54	36H
E$OUTSTANDING$CONNS	55	37H
E$ALREADY$ATTACHED	56	38H
E$DEV$DETACHING	57	39H
E$NOT$SAME$DEVICE	58	3AH
E$ILLOGICAL$RENAME	59	3BH

CODE	DECIMAL	HEXADECIMAL
E$STREAM$SPECIAL	60	3CH
E$INVALID$FNODE	61	3DH
E$PATHNAME$SYNTAX	62	3EH
E$FNODE$LIMIT	63	3FH
E$IO$UNCLASS	80	50H
E$IO$SOFT	81	51H
E$IO$HARD	82	52H
E$IO$OPRINT	83	53H
E$IO$WRPROT	84	54H
E$IO$NO$DATA	85	55H
E$IO$MODE	86	56H
E$IO$NO$SPARES	87	57H
E$IO$ALT$ASSIGNED	88	58H

D.5 CONCURRENT (PROGRAMMER ERROR) EXCEPTION CODE

CODE	DECIMAL	HEXADECIMAL
E$PARAM	32772	8004H

APPENDIX E LOGICAL DEVICES AND THE BASIC I/O SYSTEM

E.1 LOGICAL DEVICES

You can assign a logical name to any device with the I/O System call LOGICAL$ATTACH$DEVICE. This creates a logical device object, (T$LOG$DEV) and catalogs the object in the root object directory.

Typically, you use these logical device objects with I/O System calls. However, Basic I/O System calls also permit the prefix parameter to be a logical device object. When you use a logical device object as the prefix parameter in Basic I/O System calls, the Basic I/O System looks inside the logical device object to determine the device connection. In such cases, you could receive the exception code E$DEV$OFF$LINE. If you receive this exception code and the device is online, the device was never physically attached.

Before you can use a logically named device, the device must be made known to the system (attached), with the Basic I/O System call A$PHYSICAL$ATTACH$DEVICE. But when LOGICAL$ATTACH$DEVICE is invoked, the system does not immediately issue a call to A$PHYSICAL$ATTACH$DEVICE. Instead, physical attachment occurs transparently during processing of any I/O System call which references the logical device object.

You might create a logical device connection but not invoke any I/O System call to perform the physical attach operation. If so, the Basic I/O System can return E$DEV$OFF$LINE. You can correct this situation by invoking at least one I/O System call that refers to the logical device by its logical name (such as :F0:). However, your tasks must reside in an I/O Job before they can invoke I/O System calls.

E.2 REFERENCES

For further information, refer to the descriptions of I/O Jobs, LOGICAL$ATTACH$DEVICE, and A$PHYSICAL$ATTACH$DEVICE in the *Extended iRMX II Extended I/O System User's Guide*.

APPENDIX F IRMX® I AND iRMX® II BASIC I/O SYSTEM DIFFERENCES

F.1 INTRODUCTION

The iRMX II Operating System is based on the iRMX I Operating System. Therefore, the iRMX II version of the Basic I/O System is almost exactly like the iRMX I version. The same system calls are available, with no changes or additions. But there are differences. This appendix outlines the differences between the two versions for readers who are already familiar with the iRMX I Operating System. Those who aren't familiar with the iRMX I Operating System can skip this appendix.

F.2 80286 CAPABILITIES

The major differences between the iRMX I and iRMX II versions of the Basic I/O System are a result of the increased capabilities of the 80286/80386 processor--namely 16M-byte addressability and memory protection.

F.2.1 16M-Byte Addressability

The 16M-byte addressability allows device drivers (both Intel-supplied and user-written) and application tasks to access a full 16M bytes of system memory. Application tasks must use logical addresses to access memory. Logical addresses take the form

    selector:offset

On the other hand, device controllers continue to use physical addresses (16M-bytes requires a 24-bit address). Therefore your device drivers must know how to convert logical addresses to physical addresses. The *Extended iRMX II Device Drivers User's Guide* discusses this technique.

F.2.2 Memory Protection

The memory protection feature of the 80286/80386 processor protects your code and data by preventing any task from reading or writing a segment of memory unless it has explicit access to that memory segment. It also prevents memory reads or writes from crossing segment boundaries. Because of this feature, any task that uses the A$READ system call must have write access to the segment of memory used as the memory buffer. Likewise, any task that uses the A$WRITE system call must have read access to the segment used as the memory buffer. The A$SPECIAL system call also checks for the correct access and issues an error code if the access type is not appropriate for the intended operation.

F.3 DEVICE DRIVERS

Not all the Intel-supplied device drivers for iRMX I are included in the iRMX II Operating System. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for information about the Intel-supplied drivers that are available.

F.4 DISK INTEGRITY FEATURES

Three other features have been added to the Basic I/O System that have nothing to do with the capabilities of the 80286/80386 processor. All three help to improve the integrity of data stored on named volumes.

F.4.1 Attach Flags

The Basic I/O System provides an indication of the integrity of named volumes and named files. Whenever you attach a named volume or a named file, the Basic I/O System sets a flag to indicate that the volume or file is attached. The volume flag is set in the volume label; the file flag is set in the fnode file. When you detach a volume or file, the Basic I/O System clears the associated flag. Although the Basic I/O System doesn't check these flags to determine file or volume integrity, you can check the condition of a volume by invoking the A$GET$FILE$STATUS system call. If the flag is set for a volume that isn't currently attached, the setting could indicate a corrupted volume that wasn't detached properly.

The Basic I/O System doesn't provide a system call for checking the file flag. However, you can write your own programs to check this flag, or you can use the Disk Verification Utility to examine the fnode file.

F.4.2 Fnode Checksum Field

The Basic I/O System has also added a checksum field to the fnode file. The fnode file stores the information that the Basic I/O System needs to access any named file on the volume. Whenever the Basic I/O System writes the fnode file, it calculates a checksum and stores it in the fnode file. Although the Basic I/O System doesn't check the fnode file against the checksum when it reads the fnode file, your programs can use the checksum field to determine whether the fnode file has become corrupted.

F.4.3 Getting and Setting the Bad Track/Sector Information

The A$SPECIAL system call has been enhanced to retrieve and set the bad track/sector information on a volume. The bad track/sector information provides a list of the tracks or sectors on the volume that are defective. One of the new A$SPECIAL subfunctions allows you to retrieve the current list of defective tracks or sectors. The other new subfunction enables you to set up a new bad track/sector list.

A

Actions required of the reading task
  close the new file connection 5-3
  create a second file connection for the stream file 5-3
  delete the new file connection 5-4
  get the file connection 5-3
  open the new file connection for reading 5-3
  perform the required reads 5-3
Actions required of the writing task
  close the connection 5-2
  create the stream file 5-2
  delete the connection 5-2
  obtain a connection to the stream file 5-1
  open the file for writing 5-2
  pass the file connection to the reading task 5-2
  write the information to the stream file 5-2
Asynchronous system calls 6-6

B

Bad track and sector information 1-8
Basic I/O System features 1-1
  16M-byte Memory addressability 1-2
  access to files, controlling 1-5
  device independence 1-3
  file sharing 1-5
  four file types 1-4
    named files 1-4
    physical files 1-4
    remote files 1-5
    stream files 1-4
  protection features 1-2
  separation of File Lookup and File Open Operations 1-5
  support for many kinds of devices 1-3
  synchronous and asynchronous operation 1-2

BIOS performance factors 7-3
  device granularity 7-3
  file granularity 7-3
  time between updates 7-3
  volume granularity 7-3

C

Calls that return an I/O result segment C-1
Checking the condition code returned from a call C-1
Concurrent programmer errors D-3
Condition codes 6-10
Connection between tasks and device-units 2-2
Connections and physical files 4-2
Controlling fragmentation of files 1-6
Creating a file with an existing pathname 7-2

D

Data types (see Appendix A)
Device connections 2-4
Device controllers and device units 2-1
Disk integrity 1-7

E

E$IO codes returned in unit$status C-2
Environmental errors D-2
Environmental exception codes D-1
Example of using asynchronous system calls 6-7
Exception codes (see Appendix D)

F

File connections 2-5
File-path parameters for named files 6-2
Files 2-2
Flags for named volumes and files 1-8
Fnode checksum field 1-8

G

Global clock 1-7
Granularity for hard disks 1-7

I

I/O buffers for the A$READ and A$WRITE system calls 6-6
I/O result segment, structure C-2
ID, system manager 7-2
Intel devices 7-1

L

Logical devices E-1

M

Memory needed for objects B-2

N

Number of buffers for each device 7-1

O

Object types B-1
Open-mode indicator 2-7

P

Paths for system calls referring to named files 6-3
Prefix parameter1

R

Response mailbox parameter 6-4

S

Sequential programmer errors D-2
Service task priorities 7-2
Share-mode indicator 2-7
Situations requiring physical files
  formatting volumes 4-1
  implementing your own file format 4-1
  volumes formatted for other systems 4-1

Steps for using physical files
  close the file connection 4-3
  delete the file connection 4-3
  detach the device 4-3
  manipulate the file 4-3
  obtain a device connection 4-2
  obtain a file connection 4-2
  open the file connection 4-2
Structure of the I/O result segment C-2
Synchronous system calls 6-1
System initialization error reporting 7-2
System manager 7-2

T

The two parts of asynchronous system calls
  the concurrent part 6-6
  the sequential part 6-6
Three types of granularity
  device 1-6
  file 1-6
  volume 1-6
Timing facilities 7-2

U

Unit status errors for the iSBC 214/215G Controller C-3
User parameter 6-2
Using GET$TYPE to determine if a connection or an I/O Result Segment has been returned C-1
Using physical files 4-2

V

Volume granularity for flexible diskettes 1-6
Volumes 2-2

EXTENDED iRMX®II EXTENDED I/O SYSTEM USER'S GUIDE

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

intel® PREFACE

INTRODUCTION

This manual describes the iRMX® II Extended I/O System. Although it contains some introductory and overview material, it is primarily a detailed description of the Extended I/O System.

READER LEVEL

The manual is written for programmers who are already familiar with

- The concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide.*
- The PL/M-286 programming language.

Readers need not be familiar with the iRMX II Basic I/O System.

MANUAL ORGANIZATION

This manual is divided into the following chapters and appendixes.

Chapter 1	This chapter describes the differences between the Basic I/O System and the Extended I/O System. You should read this chapter if you are not certain which I/O system best meets your requirements.
Chapter 2	This chapter describes the primary features of the Extended I/O System. You will find this chapter particularly useful if you have not used the Extended I/O System before.
Chapter 3	This chapter explains some basic terminology associated with the Extended I/O System, including the following terms: device, volume, file, and connection. You should read this chapter if you are looking through the manual for the first time or if you are unfamiliar with the Extended I/O System.
Chapters 4-6	These chapters describe named, physical, remote, and stream files and how to use them. You should read one or more of these chapters, depending on the kinds of files your application uses.

Chapter 7	This chapter describes characteristics of the Extended I/O System that you specify when you configure an iRMX II Operating System.
Appendix A	This appendix defines the formats of the data types used by the Extended I/O System. For example, it explains the format of an iRMX II STRING.
Appendix B	This appendix provides a list of the types of objects created by the Extended I/O System. It also discusses the resource requirements of the Extended I/O System.
Appendix C	This appendix contains a list of the condition codes that the Extended I/O System can return. The codes are listed in alphabetical order, and each entry in the list includes the classification of the code (programmer error or environmental condition) and the numeric value of the code.
Appendix D	The Extended I/O System uses object directories extensively. This appendix tells which entries are used by the Extended I/O System. It also tells you which entries you can change and which entries you can't.
Appendix E	This appendix explains the incompatibilities between the system calls of the Basic I/O System and the system calls of the Extended I/O System.
Appendix F	This appendix lists the differences between the iRMX II and iRMX I versions of the Extended I/O System.

CONVENTIONS

This manual uses the following conventions:

- The term "iRMX II" refers to the Extended iRMX II.3 Operating System.
- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- All iRMX II system calls begin with one of two standard prefixes: RQ$ or RQE$. When referring to the system calls that begin with RQ$, this manual uses a shorthand notation and omits the prefix. For example, S$CREATE$FILE means RQ$S$CREATE$FILE. The actual PL/M-286 external procedure names used to invoke these system calls are shown only in the *Extended iRMX II Extended I/O System Calls Reference Manual*, which lists the detailed calling sequences.

- When referring to system calls that begin with RQE$, this manual spells out the complete names, including the RQE$ characters.
- There are some system calls whose names are identical except for the RQ$ or RQE$ prefix (for example, the EIOS system calls RQ$CREATE$IO$JOB and RQE$CREATE$JOB). The difference between two similarly named system calls is that the RQ$ version operates as it did under the iRMX I Operating System and is available for compatibility. The RQE$ version is updated to support new 80286 features, such as 16M byte memory pools. Unless compatibility with iRMX I systems is an issue, Intel recommends that you use the system call with the RQE$ preface instead of the one with the RQ$ preface.

CONTENTS

CHAPTER 1 — CHOOSING BETWEEN I/O SYSTEMS PAGE

1.1 Overview ... 1-1
1.2 Reason For Having Two I/O Systems ... 1-1
  1.2.1 Basic I/O System ... 1-2
  1.2.2 Extended I/O System ... 1-3
1.3 Making The Decision ... 1-3
  1.3.1 Memory Requirements ... 1-4
  1.3.2 Performance ... 1-4
1.4 Examples ... 1-4
  1.4.1 Application Systems Using Little I/O ... 1-4
  1.4.2 Application Systems Using Only Sequential I/O ... 1-5
  1.4.3 High Performance Applications Using Random I/O ... 1-5
1.5 Summary ... 1-6

CHAPTER 2 — FEATURES OF THE EXTENDED I/O SYSTEM PAGE

2.1 Introduction ... 2-1
2.2 16M-Byte Memory Addressability ... 2-1
2.3 Protection Features ... 2-2
2.4 Support For Many Kinds Of Devices ... 2-2
2.5 Device Independence ... 2-2
2.6 Four Kinds Of Files ... 2-3
  2.6.1 Named Files ... 2-3
  2.6.2 Physical Files ... 2-3
  2.6.3 Stream Files ... 2-3
  2.6.4 Remote Files ... 2-3
2.7 File Independence ... 2-4
2.8 Separation Of File Lookup And File Open Operations ... 2-4
2.9 File Sharing And Access Control ... 2-4
  2.9.1 File Sharing ... 2-5
  2.9.2 Access Control ... 2-5
2.10 Buffering With Overlapped I/O ... 2-5
  2.10.1 Advantages Of Using Buffers ... 2-5
  2.10.2 Buffer Size ... 2-6
2.11 Logical Names For Files And Devices ... 2-7
2.12 Automatic Reattachment Of Devices ... 2-7

CHAPTER 3 PAGE
FUNDAMENTAL CONCEPTS
3.1 Introduction 3-1
3.2 Device Controllers And Device Units 3-1
3.3 Volumes 3-1
3.4 Files 3-2
3.5 Communication Connections Between Tasks And Device Units 3-2
3.5.1 Interlayer Bonds Preceding Initialization 3-3
3.5.2 Post-Initialization Bond - the Configuration Interface 3-4
3.5.2.1 Device Connections 3-5
3.5.2.2 File Connections 3-5
3.5.2.3 File Pointers 3-6
3.5.2.4 Some Observations about Devices and Connections 3-6
3.6 Logical Names 3-7
3.6.1 Syntax for Logical Names 3-7
3.6.2 Logical Names and Object Directories 3-7
3.7 I/O Jobs 3-8
3.8 Path$Ptr Parameters And Default Prefixes 3-9

CHAPTER 4 PAGE
NAMED FILES
4.1 Overview 4-1
4.2 Multiple Files On A Single Volume 4-1
4.3 Hierarchical Naming Of Files 4-2
4.3.1 System Calls Requiring Connections 4-4
4.3.2 System Calls Requiring Paths 4-5
4.3.2.1 Prefixes 4-5
4.3.2.2 Subpaths 4-6
4.3.2.3 Using Prefixes in Conjunction With Subpaths 4-6
4.3.2.4 Specifying Paths in System Calls 4-6
4.3.2.5 Path Syntax 4-6
4.4 Controlling Access To Files 4-8
4.4.1 Users and User Objects 4-8
4.4.1.1 User IDs 4-9
4.4.1.2 User Objects 4-9
4.4.1.3 Default User Object for a Job 4-9
4.4.1.4 Supporting Dynamic Logon and iRMX-NET 4-10
4.4.2 Types of Access to Files 4-10
4.4.3 File Access List 4-11
4.4.4 Computing Access for File Connections 4-12
4.4.5 Special Users 4-14
4.4.5.1 System Manager User 4-14
4.4.5.2 World User 4-14

CHAPTER 4 (continued) PAGE
4.5 Extended I/O System Calls For Named Files 4-15
4.5.1 Obtaining and Deleting Connections 4-15
4.5.2 Manipulating Data 4-16
4.5.3 Obtaining Status 4-17
4.5.4 Deleting and Renaming Files 4-17
4.5.5 Changing Access 4-17
4.5.6 Deleting Connections 4-17
4.5.7 Using Logical Names 4-17
4.5.8 Creating and Deleting I/O Jobs 4-18
4.5.9 Performing Miscellaneous Functions 4-18
4.6 Basic I/O And Nucleus System Calls 4-19
4.7 Chronological Overview Of Named Files 4-19

CHAPTER 5
PHYSICAL FILES PAGE
5.1 Overview 5-1
5.2 Situations Requiring Physical Files 5-1
5.3 Connections And Physical Files 5-2
5.4 Suggestion For Maintaining File Independence 5-2
5.5 Using Physical Files 5-2

CHAPTER 6
STREAM FILES PAGE
6.1 Overview 6-1
6.2 Suggestion For Maintaining File Independence 6-1
6.3 Stream File Protocols 6-1
6.3.1 Protocol for the Creating Task 6-2
6.3.2 Protocol for the Writing Task 6-2
6.3.3 Protocol for the Reading Task 6-3

CHAPTER 7
CONFIGURATION OF THE EXTENDED I/O SYSTEM PAGE
7.1 Overview 7-1
7.2 I/O Job Objects 7-1
7.3 Internal Buffer Size 7-1
7.4 I/O Job Characteristics 7-2
7.5 Automatic Boot Device Recognition 7-2
7.6 Initialization Error Reporting 7-3

APPENDIX A
DATA TYPES PAGE
A.1 Data Types A-1

APPENDIX B PAGE
OBJECT TYPES AND RESOURCE REQUIREMENTS
B.1 Overview....B-1
B.2 Object Types....B-1
B.3 RAM Requirements....B-2
B.3.2 Creating an I/O Job....B-2
B.3.3 Opening a Connection....B-3
B.3.4 Other RAM Requirements....B-3
B.4 Object Counts....B-3

APPENDIX C PAGE
CONDITION CODES
C.1 Overview....C-1
C.2 Normal Condition Code....C-1
C.3 Programming Errors....C-2
C.4 Environmental Conditions....C-2
C.4 Environmental Conditions....C-3

APPENDIX D PAGE
USE OF OBJECT DIRECTORIES BY THE EXTENDED I/O SYSTEM
D.1 Object Directories....D-1

APPENDIX E PAGE
COMPATIBILITY BETWEEN THE TWO I/O SYSTEMS
E.1 Compatibility Between The Two I/O Systems....E-1

APPENDIX F PAGE
iRMX® I AND iRMX® II EXTENDED I/O SYSTEM DIFFERENCES
F.1 Introduction....F-1
F.2 Extended Memory Pool Sizes....F-1
F.3 Protection Features....F-1
F.4 New System Calls....F-2

TABLES

TABLE PAGE

B-1 Type Codes B-1

FIGURES

FIGURE PAGE

3-1 Layers of Interfacing Between Tasks and a Device 3-2
3-2 Schematic of Software at Initialization Time 3-3
3-3 A System with Device and File Connections 3-4
4-1 Example of a Named File Tree 4-3
4-2 Computing the Access Mask for a File Connection 4-13
4-3 Chronology of Frequently Used System Calls for Named Files 4-20

CHAPTER 1 CHOOSING BETWEEN I/O SYSTEMS

1.1 INTRODUCTION

The iRMX II Operating System provides you with a choice of two I/O systems. One of these, the Extended I/O System, is described in this manual. The other, the Basic I/O System, is described in the *Extended iRMX II Basic I/O System User's Guide.*

This chapter explains the reason for having two I/O systems and briefly describes the differences between them. After reading this chapter, you should be able to decide whether your application requires system calls from both systems or from just one. If one of the I/O systems is sufficient, you should be able to decide which one.

As you read this chapter, remember that the Extended I/O System is built upon the Basic I/O System. In other words, if you choose the Extended I/O System, you must also include the Basic I/O System in your application.

1.2 REASON FOR HAVING TWO I/O SYSTEMS

The iRMX II Operating System is designed to provide Original Equipment Manufacturers (OEMs) with a variety of features that are useful in building application systems. Many of these features are useful in most, but not all, applications. This is especially true of features relating to input and output.

Most applications communicate with external devices such as line printers, terminals, disk drives, or bubble memories. Even so, not all applications have the same requirements. The iRMX II Operating System provides two I/O systems to allow you to choose the one that best satisfies the requirements of your application system. And, in the event that both systems would prove useful in one application, the iRMX II Operating System allows you to use them both.

NOTE

All information on iRMX Networking Software (iRMX-NET) can be found in the *iRMX Networking Software User's Guide*. Consult this manual for information on how to share files and data in a distributed environment with separate iRMX-based workstations. The *iRMX Networking Software User'g Guide* is not part of the iRMX II documentation set.

1.2.1 Basic I/O System

The Basic I/O System is the more flexible of the two I/O systems. It provides very powerful capabilities, and it makes few assumptions about the requirements of your application. The following features illustrate the flexibility of the Basic I/O System:

- ALLOWS YOU TO DESIGN YOUR OWN BUFFERING ALGORITHM

  Although many applications require buffered I/O, the Basic I/O System does not automatically provide it. Rather than require one particular approach to buffered I/O for all applications, the Basic I/O System allows you to design and implement your own buffering method.

- SUPPLIES ASYNCHRONOUS SYSTEM CALLS

  Rather than making assumptions about whether (and how) you wish to overlap your I/O operations, the Basic I/O System allows you to explicitly control the synchronization of the system calls.

- GIVES YOUR TASK CONTROL OF DETAILS

  The system calls of the Basic I/O System involve many parameters. Using these parameters, your tasks can closely tailor the behavior of each system call to match the requirements of your application system.

As these features show, the Basic I/O System emphasizes flexibility rather than ease of use. By preserving flexibility, the Basic I/O System provides I/O features that are useful in a wide range of applications.

Clearly, the Basic I/O System does have limitations. Many applications that perform I/O do not need the control of details afforded by the Basic I/O System. For many applications, the amount of time required to develop the application system is more critical than the ability to finely tune its performance. For these applications, the iRMX II Operating System provides the Extended I/O System.

1.2.2 Extended I/O System

The Extended I/O System is designed to be easier to use than the Basic I/O System. The following features of the Extended I/O System help make it easier to use:

- AUTOMATIC BUFFERING OF I/O OPERATIONS

  The Extended I/O System provides you with automatic buffering of all I/O operations. Aside from specifying how many buffers the Extended I/O System uses, your tasks need not become involved with buffering. Furthermore, if your application system does not require buffering, your tasks can tell the Extended I/O System to use no buffers.

- SYNCHRONOUS SYSTEM CALLS

  The Extended I/O System provides system calls that are synchronous. By freeing your application software from the burden of explicitly synchronizing system calls, the Extended I/O System reduces the complexity of your application system. This, in turn helps reduce development costs. Although the system calls of the Extended I/O System are synchronous, your application system can still use overlapped I/O operations. To do so, your tasks need only tell the Extended I/O System to use buffers, and the Extended I/O System will automatically overlap your I/O operations.

- FREES YOUR TASKS OF TEDIOUS DETAILS

  The system calls of the Extended I/O System require fewer parameters than do those of the Basic I/O System. This simplifies your application system and reduces development costs.

1.3 MAKING THE DECISION

At this point, you are faced with three alternatives. Should you use the Basic I/O System, the Extended I/O System, or both? To make this decision, decide if your application system requires the flexibility and fine tuning of the Basic I/O System, the ease of use of the Extended I/O System, or a combination of both. Before you make the final decision, consider the following two factors.

1.3.1 Memory Requirements

The Extended I/O System software requires the Basic I/O System. The size of the Extended I/O System software is approximately 19K bytes. If your application system is pressed for memory and does not require features of the Extended I/O System, you can save the 19K bytes of memory by using only the Basic I/O System.

However, if you decide to use the Basic I/O System even though your application system needs the features of the Extended I/O System (such as buffering), you might end up using all 19K bytes of memory implementing these same features on top of the Basic I/O System.

Be aware that using both the Basic I/O and Extended I/O Systems requires no more memory than using the Extended I/O System alone.

1.3.2 Performance

Because the Basic I/O System gives your application system control of many details, you can probably tune your application system to run faster with the Basic I/O System than with the Extended I/O System. So if performance is more important than reduced development costs, you should consider using the Basic I/O System. If you decide to use the Extended I/O System, you can improve performance by changing the buffer sizes.

1.4 EXAMPLES

The following examples illustrate the advantages of each of the I/O systems. The analysis in each example is based on the assumption that many copies of the application system are to be produced.

1.4.1 Application Systems Using Little I/O

Suppose that your application system requires very little I/O. For instance, suppose that its only I/O activity is to occasionally log information to a flexible disk.

Because this application system involves very few I/O-related system calls, the Basic I/O System is preferable to the Extended I/O System. The ease of use provided by the Extended I/O System can save you very little manpower (hence money and time) during development because the I/O-related part of your system requires so little time to develop. This marginal benefit is of less use to your application system than is the 19K bytes of memory saved by using the Basic I/O System.

1.4.2 Application Systems Using Only Sequential I/O

Suppose that your application system requires a substantial amount of sequential I/O. In this type of system, a large amount of your development resources will be expended in support of I/O. This factor should make you consider using the Extended I/O System to reduce your staffing requirements while developing the application system.

A second factor should also steer you toward the Extended I/O System--the sequential I/O. The buffering scheme used by the Extended I/O System is particularly efficient while performing sequential I/O because it incorporates read-ahead and write-behind algorithms to overlap I/O operations and processing.

These two factors, the amount of manpower required to implement I/O and the sequential nature of the I/O, combine to make the Extended I/O System the best choice for this application system.

1.4.3 High Performance Applications Using Random I/O

Now suppose that your system performs a large amount of random-access I/O, and suppose that performance is a critical consideration that overrides concerns about conserving memory. You should consider the Extended I/O System because, in this application system, it can substantially reduce your development costs. However, two other factors combine to make the Basic I/O System another reasonable choice.

The first factor is the random nature of the I/O. The read-ahead and write-behind algorithm provided by the Extended I/O System is not particularly efficient in random-access I/O operations. The second factor is the requirement for performance. Using the Basic I/O System as a foundation, you can build an I/O facility that takes advantage of your application system's knowledge of the organization of data in the files. Although such a facility might be expensive to implement, it should run faster than the Extended I/O System in this application.

So in this case, you must weigh the cost of development against the benefit of better performance. If development costs are more important, you should use the Extended I/O System. If performance is more important, you should use the Basic I/O System. Also, don't ignore the option of using the Extended I/O System to create a prototype application system and then later replacing the Extended I/O System with your custom I/O facility.

1.5 SUMMARY

In general, you should consider the Basic I/O System for applications that require very little I/O, or for applications requiring finely-tuned performance while doing random-access I/O. In contrast, you should consider the Extended I/O System when development costs are critical, especially in applications that use sequential I/O.

Finally, remember that there are circumstances where you should use both I/O systems. One such situation occurs when your application system uses I/O for several purposes, some of which are best accomplished by the Basic I/O System, and others by the Extended I/O System.

CHAPTER 2 FEATURES OF THE EXTENDED I/O SYSTEM

2.1 INTRODUCTION

Because the Extended iRMX II Extended I/O System is designed primarily for use by Original Equipment Manufacturers (OEMs), it provides a large number of features--including some that are not generally found in operating systems aimed at end users. These features include

- Memory protection
- Support for many kinds of devices
- Device independence
- Four distinct kinds of files
- File independence
- Separation of file-lookup and file-open operations
- File sharing and access control (support for iRMX-NET)
- Buffering with overlapped I/O
- Logical names for files and devices
- Automatic reattachment of devices
- 16M-byte memory addressability

The first eight of these features are implemented by the Basic I/O System. However, because the Extended I/O System uses the facilities provided by the Basic I/O System, the Extended I/O System also provides these features. The balance of the features are available only with the Extended I/O System.

2.2 16M-BYTE MEMORY ADDRESSABILITY

The iRMX II Operating System runs in the protected virtual address mode (PVAM) of the 80286 and 80386 processors. As a result, it can access as much as 16M bytes of memory. The Extended I/O System takes advantage of this feature by allowing you to create I/O jobs with memory pools of up to 16M bytes. Therefore, tasks that invoke Extended I/O System calls can have more code and can have more room for data.

2.3 PROTECTION FEATURES

Because the iRMX II Operating System accesses the processor in PVAM, it benefits from some of the inherent memory protection features of the processor. These features protect your code and data by preventing any task from reading or writing buffers of memory unless it has explicit access to those buffers. They also prevent memory reads or writes from crossing segment boundaries. The Operating System generates exception codes if an attempted protection violation occurs.

The Operating System also checks system call parameters for protection violations and for incorrect values. Appendix C lists the exception codes that can be returned.

2.4 SUPPORT FOR MANY KINDS OF DEVICES

The Extended I/O System supports a wide variety of devices. To connect a particular device to the Extended I/O System, your application system must include a device driver (a collection of software procedures implemented at the Basic I/O System level) for the device being connected. The Operating System provides you with drivers for flexible and hard disks, line printers, serial terminals, bubble memories, and many other devices. A complete list of devices that are supported by the Operating System is included in the *Extended iRMX II Interactive Configuration Utility Reference Manual*, along with detailed instructions for including specific drivers in your application system.

If you need drivers for devices other than those for which Intel supplies drivers, you can write drivers that are compatible with the iRMX II Operating System. For specific instructions, refer to the *Extended iRMX II Device Drivers User's Guide.*

2.5 DEVICE INDEPENDENCE

The Extended I/O System provides you with one set of system calls that can be used with any collection of devices. For instance, rather than using a TYPE system call for output to a terminal and a PRINT system call for output to a line printer, you can use a WRITE system call for output to any device.

This notion of one set of system calls for I/O to any collection of devices is called device independence, and it provides your application with a lot of flexibility. For example, suppose your application logs events as they occur. The device independence of the Extended I/O System allows you to create an application that can log the events on any device rather than on just one. When the application is running and circumstances force an operator to reroute logging from the teletypewriter to the line printer or disk, your application can easily comply. For a more detailed explanation of device independence, refer to the *Introduction to the Extended iRMX II Operating system.*

2.6 FOUR KINDS OF FILES

The Basic I/O System implements four distinct kinds of files, and the Extended I/O System supports all four. Each kind of file is byte-oriented, rather than record-oriented.

2.6.1 Named Files

Named files are intended for use with random-access, secondary-storage devices such as disk drives, diskette drives, and bubble memories. Named files allow your application to organize its files into a tree-like, hierarchical structure that reflects the relationships between the files and the application. Furthermore, only named files allow your application to store more than one file on a device, and only named files provide your application with access control. Named files also provide you with a good foundation for building custom access methods such as ISAM (indexed sequential access method).

For more detailed information regarding named files, read Chapter 4 of this manual.

2.6.2 Physical Files

Physical files differ from named files in that physical files allow your application more direct control over a device. Each physical file occupies an entire device, and applications can deal with the file as though it were a string of bytes. However, physical files do not provide access control. This more-basic relationship with a device provides your application with flexibility. For example, your application can use a physical file to interpret volumes created on other systems.

Physical files also provide your application with the ability to communicate with devices that do not need the power of named files. Several examples of such devices are line printers, display tubes, plotters, and robots.

2.6.3 Stream Files

Stream files provide a means for two tasks to communicate with each other. One task writes into the file while the other task concurrently reads from it. Stream files use no devices and provide no access control.

2.6.4 Remote Files

The Extended I/O System can also access remote files through OpenNET. For more information on accessing remote files, consult the *iRMX Networking Software User's Guide*.

2.7 FILE INDEPENDENCE

System calls for reading and writing work with any of the types of files described earlier. This allows you to create tasks and applications that can be readily switched from one kind of file to another.

For example, your application might involve two tasks that must communicate by using a stream file. In the process of developing the application system, you might implement the writing task before you implement the reading task. For the purpose of debugging the writing task, you can use a named file on a disk so you can examine the information being written. Later, after you implement the reading task, you can route the information to the stream file rather than the disk.

2.8 SEPARATION OF FILE LOOKUP AND FILE OPEN OPERATIONS

Many operating systems waste valuable time by looking up a file whenever an application tries to open one. The iRMX II Extended I/O System avoids this overhead by using a special type of iRMX II object (called a file connection) to represent the bond between the file and an application program.

Whenever your application software creates a file, the iRMX II Extended I/O System returns a file connection. Your application can then use the connection to open the file without suffering the expense of having the Extended I/O System look up the file. Even when your application opens an existing file, the application can present the file connection and bypass the file-lookup process.

File connections provide a second benefit, one that relates to access control. Any connection to a named file embodies the access rights to the file. This means that the Extended I/O System computes access only once (when the file connection is created), rather than each time the file is opened.

Another benefit of file connections is that several of them can simultaneously exist for the same file. This allows several tasks to concurrently access different locations in the file. This is possible because each file connection maintains a pointer to keep track of the location within the file where the task is reading or writing.

If you plan to access remote files through OpenNET, consult the *iRMX Networking Software User's Guide*. That manual explains how to access iRMX II-based files that reside on remote hardware.

2.9 FILE SHARING AND ACCESS CONTROL

The Extended I/O System provides your application with the ability to share files and, in the case of named files, to control access to files.

2.9.1 File Sharing

In a multitasking system, it is often useful to have several tasks manipulating a file simultaneously. For example, consider a transaction processing system in which a large number of operators concurrently manipulate a common data base. If each terminal is driven by a distinct task, the only way to implement an efficient transaction system is to have the tasks share access to the data-base file. The Extended I/O System allows multiple tasks to access the same file at the same time.

2.9.2 Access Control

Also useful in a multitasking system is the ability to control access to a file. For instance, suppose that several engineering departments share a computer. An engineer in one department may want to control access to files as follows:

- Allow the ability to read, write, and delete files.
- Allow other engineers within the department to read and write the files, but deny them permission to delete the files.
- Allow engineers of other departments to only read the files.

Named files provide your application with precisely this kind of access control.

For more detailed information regarding access control, read Chapter 4 of this manual.

2.10 BUFFERING WITH OVERLAPPED I/O

The Extended I/O System provides buffering and overlapping of I/O operations.

2.10.1 Advantages Of Using Buffers

Whenever one of your application programs opens a connection, the program must specify the number of buffers to be provided by the Extended I/O System. The number of buffers that your program requests affects the actions of the Extended I/O System as it reads and writes information through the connection. Specifically

- Zero Buffers

  The Extended I/O System actually accesses the file each time your application invokes a read system call or a write system call. For example, if your application code asks the Extended I/O System to read 30 bytes, the Extended I/O System accesses the file and reads exactly 30 bytes. If the file resides on a physical device, such as a disk, the Extended I/O System accesses the file for each read or write request.

- One Buffer

  The Extended I/O System reads and writes information one buffer at a time. For instance, when your application program asks the Extended I/O System to read 30 bytes, the System instead reads enough information to fill the entire buffer. Using this method, the Extended I/O System might be able to satisfy several additional requests without reading the file again.

  This method of transferring a full buffer at a time is called blocking. Blocking can significantly improve the performance of an application system by reducing the number of times the Extended I/O System must actually transfer information to or from a file on a device. In general, blocking is more valuable in sequential I/O than in random I/O.

- Two or More Buffers

  If your application requests two or more buffers, the Extended I/O System can overlap I/O operations by using read-ahead and write-behind algorithms.

  Reading ahead and writing behind are techniques for allowing tasks to continue running while the Extended I/O System is transferring information to or from devices. Both techniques are particularly useful when your application is performing sequential (rather than random-access) I/O. This is because the Extended I/O System can accurately determine, during sequential reading or writing, the location of the next data required by the application.

  When you configure the Operating System, you specify the maximum number of buffers that the Extended I/O System can use for files on a particular device. The number of buffers that the Extended I/O System actually uses when reading or writing a file is the lesser of this maximum value and the number of buffers specified when the file is opened. The S$OPEN call is described in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

2.10.2 Buffer Size

If you are responsible for configuring your application system, you should be aware that buffer sizes are, at least partially, a function of some parameters that you set during the configuration process. The next two paragraphs discuss these parameters. However, if you are not involved with configuration of your system, you can skip over these paragraphs without missing any crucial information.

When your application requests one or more buffers, the Extended I/O System computes the size of the buffers as a function of two configuration parameters--the granularity of the device, and the internal buffer size of the Extended I/O System. The granularity is a Basic I/O System configuration parameter, and the internal buffer size is an Extended I/O System configuration parameter. For more information about configuration, refer to Chapter 7.

When your application program opens a connection, the Extended I/O System creates buffers equal to the largest integral multiple of the device granularity that does not exceed the internal buffer size. There are two exceptions to this rule:

- If the device granularity is zero, the Extended I/O System creates buffers equal to the internal buffer size.
- If the device granularity is greater than the internal buffer size, the Extended I/O System creates buffers equal to the internal buffer size.

2.11 LOGICAL NAMES FOR FILES AND DEVICES

The Extended I/O System allows your application program to use logical names to refer to files and devices. A logical name is a string of characters that the Extended I/O System associates with a particular file connection or device connection. (A device connection relates to devices in the same way that a file connection relates to files. Refer to Chapter 3 of this manual for a precise definition.)

You can make a logical name available to one job, to a group of jobs, or to all jobs in the system. This is accomplished by cataloging the name in the local job object directory, the global job object directory, or the root job's object directory, respectively. (Chapter 3 describes these directories.)

2.12 AUTOMATIC REATTACHMENT OF DEVICES

The Extended I/O System constantly monitors the status of devices. When an operator removes a diskette (or any other removable media) from a drive that is capable of detecting a volume being removed, the Extended I/O System detaches the device and deletes all connections to files on the device. When the operator replaces the removed media, the Extended I/O System automatically re-attaches the device as soon as it is accessed, making it available to the tasks in your system.

Some devices, such as some 5.25-inch flexible diskette drives, cannot detect a volume being removed from the drive. For these devices, the Extended I/O System cannot perform automatic reattachment.

CHAPTER 3
FUNDAMENTAL CONCEPTS

3.1 INTRODUCTION

Before you use the Extended I/O System, you should understand several fundamental concepts. Some of those concepts were presented in Chapter 2. The remaining concepts are

- Device Controllers and Device Units
- Volumes
- Files
- Connections
- I/O Jobs
- Logical Names

The following sections explain these concepts.

3.2 DEVICE CONTROLLERS AND DEVICE UNITS

Devices are such things as flexible diskette drives, line printers, terminals, card readers, and the like. A device is a hardware entity that tasks can use to read information, write information, or do both. More precisely, these are device units.

The iRMX II Operating System distinguishes between device units and the hardware interfaces that control device units. The latter interfaces are called device controllers. Typically, a device controller allows iRMX II application software to communicate with several device units. For example, an iSBC 214 Winchester Disk Controller board acts as an interface between application software and several Winchester disk drives (device units).

3.3 VOLUMES

A volume is the medium used to store the information on a device unit. For example, if the device unit is a flexible disk drive, the volume is a diskette; and if the device unit is a multi-platter hard disk drive, the volume is the disk pack.

3.4 FILES

Some operating systems treat a file as a device, while others treat a file as information stored on a device. The Extended I/O System considers a file to be information.

The Extended I/O System provides four kinds of files: physical, named, stream, and remote. Each kind has characteristics that make it unique. Chapter 2 provides general information about each kind of file. Chapters 4, 5, and 6 provide more detail about named, physical, and stream files. The *iRMX Networking Software User's Guide* provides detailed information about remote files.

Regardless of the kind of file, the Extended I/O System provides information to applications as a string of bytes, rather than as a collection of records.

3.5 COMMUNICATION CONNECTIONS BETWEEN TASKS AND DEVICE UNITS

In complex environments, such as those supported by the iRMX II Operating System, several layers of software and hardware must be bound together before communication between application tasks and device units can commence. Figure 3-1 shows these layers.

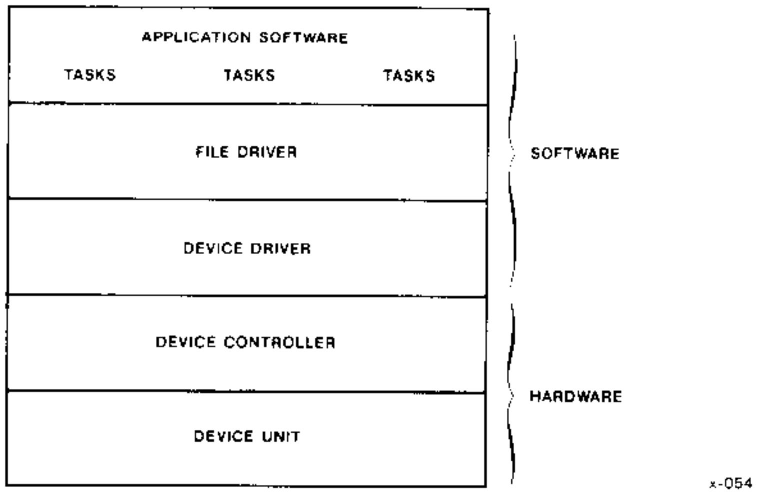


Figure 3-1. Layers of Interfacing Between Tasks and a Device

3.5.1 Interlayer Bonds Preceding Initialization

The bond between a device controller and the device units that it controls is a physical bond, usually in the form of wires or cables.

A device driver is bound to device controllers by data residing in a data structure known as a Device Unit Information Block (DUIB). You supply the data for DUIBs when configuring the Operating System. Refer to the *Extended iRMX II Device Drivers User's Guide* for more information about DUIBs.

Application software is bound to the file driver during the linking process. You supply the information needed to perform the binding process when configuring the system.

When your application starts up, these three bonds are in place, leaving only one gap between the layers. Figure 3-2 illustrates this situation. The new element, shown in the figure as the configuration interface, is the "glue" that provides the final bond.

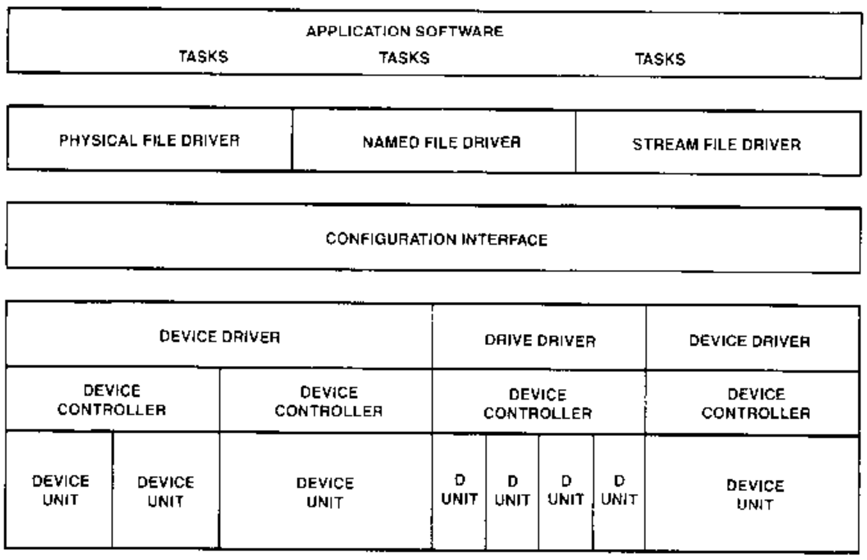


Figure 3-2. Schematic of Software at Initialization Time

3.5.2 Post-Initialization Bond - the Configuration Interface

The configuration interface provides two kinds of system calls. Before a task can use a file, both of these kinds of calls must be invoked. One type of call produces a device connection, and the other a file connection. These two types of connections are shown in Figure 3-3. Device connections are depicted as conduits (pipes); file connections are shown as wires through the conduits.

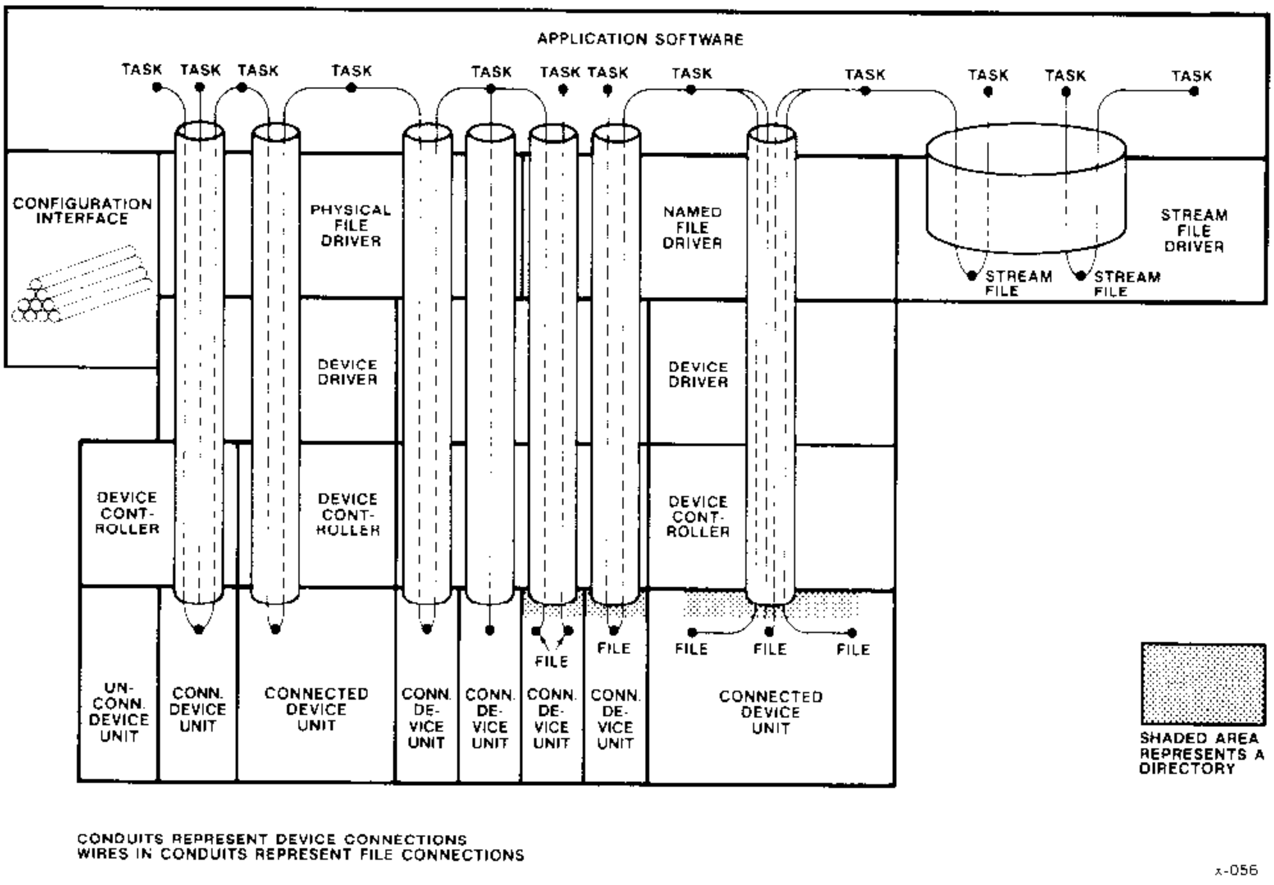


Figure 3-3. A System with Device and File Connections

3.5.2.1 Device Connections

To use a device for Extended I/O System calls, tasks must first invoke the system call LOGICAL$ATTACH$DEVICE, which

- Creates an object type called a logical device object--also referred to as a device connection--that represents the device.
- Catalogs a token for the new object under a logical name specified in the LOGICAL$ATTACH$DEVICE call. (A later section in this chapter discusses logical names in detail.)
- Identifies the owner of the device connection, to prevent other users from detaching devices that they do not own.

After LOGICAL$ATTACH$DEVICE is invoked, the logical name is associated in the object directory with a token for the logical device object. The Extended I/O System accesses the device by using this logical name.

3.5.2.2 File Connections

When an application task accesses a device unit, it must use the logical name for that device unit to obtain a file connection object. The file connection represents a particular file on a device unit. How the task obtains the connection depends on whether the file already exists. If the file already exists, the task usually calls S$ATTACH$FILE, although it can also call S$CREATE$FILE. If the file does not yet exist, the task must call S$CREATE$FILE.

NOTE

You can call S$CREATE$FILE to obtain a file connection for a file that already exists. But, for the following reason, you should avoid using S$CREATE$FILE unless it is certain that the file does not yet exist.

Calling S$CREATE$FILE to obtain a connection for a file that already exists truncates the file to zero length. Even if you do this deliberately, truncating causes problems for tasks having other connections to the file, because the file pointers (discussed later in this section) for those other connections are not affected, even though the end-of-file marker is moved to the beginning of the file.

You should observe this precaution even when obtaining connections to physical or stream files, because you might want to use the same code to obtain connections to named files.

Unlike device connections, there can be multiple file connections to one file. This allows different tasks to have different kinds of access to the same file at the same time, as the next paragraph shows.

After receiving a file connection, a task calls S$OPEN to open the connection. Parameters in the call to S$OPEN specify how the task intends to use the file connection and how it is willing to share the file with other tasks.

NOTE

If a task in one job obtains a file connection that was created in a different job, the task cannot successfully use the connection to perform I/O operations. However, the task can catalog the connection under a logical name, and use the logical name in the ATTACH$FILE system call to obtain a second connection that can be used without restriction.

3.5.2.3 File Pointers

The Extended I/O System maintains a file pointer for each open file connection to a random-access device unit. This file pointer tells the Extended I/O System the logical address of the byte where the next I/O operation on the file is to begin. The logical addresses of the bytes in a file begin with zero and increase sequentially through the entire file. Normally the pointer for a file connection points at the next logical byte after the one most recently read or written. However, a task can modify the file pointer by invoking the S$SEEK system call. This seeking process is particularly useful when performing random-access (as opposed to sequential) operations on a file.

3.5.2.4 Some Observations about Devices and Connections

Figure 3-3 is quite detailed and shows most of the situations that are possible for device units and file connections to them. In particular, you can observe the following about the figure:

- Device connections extend from the application software to the individual device units, and each passes through one and only one file driver.
- There is only one device connection to each connected device. However, multiple file connections can share the same device connection.
- Different device units with the same controller can be connected via different file drivers.
- Tasks can share access to the same device unit through the physical file driver, and they can share access to the same files on the same device unit through the named file driver.
- There is only one device connection through the stream file driver, because one logical device contains all stream files.
- The configuration interface, which is depicted as a pile of conduits, is off to one 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.

3.6 LOGICAL NAMES

Logical names identify file connections or device connections. More specifically, you catalog the tokens for file connections or device connections under logical names. This section describes the syntax for logical names, and then describes the directories in which you catalog connections under logical names.

3.6.1 Syntax for Logical Names

A logical name is a STRING of 12 or fewer characters with the following characteristics:

- The ASCII code for each character must be between 020h and 07Fh. You cannot use a slash (/), an up-arrow (↑), or a circumflex (^) in the logical name. A logical name may be enclosed in colons, (for example, :f0:), but the colon character cannot be used as part of the logical name. If the logical name is to be used as a prefix to a pathname, it must be enclosed in colons. (Prefixes are discussed in a later section of this chapter.)
- Leading and embedded blanks are significant. For example, if the STRING used to define a logical name contains a leading blank followed by two X's, the logical name is not simply two X's. Rather it is a blank followed by two X's.
- The Extended I/O System does not distinguish between uppercase and lowercase letters in logical names. For example, the Extended I/O System considers xyz and XYZ to be the same logical name.

3.6.2 Logical Names and Object Directories

Every I/O job has three distinct types of object directories in which objects can be cataloged. (The specific characteristics of an I/O job are described in the next section.)

When looking up a logical name, the Extended I/O System searches these directories in the order they are listed here, and it stops when it finds the name. One effect of this search scheme is that you can make a logical name available to only one job, to a group of jobs, or to all jobs in the system.

The object directories searched, and the order in which they are searched, are

1. The object directory of the local job. The local job is the job containing the task that requested the Extended I/O System to find the logical name.

   If you wish to share a connection with tasks in the same job, but not other jobs, catalog the token for the connection under a logical name in the local object directory.

2. The object directory of the global job. An I/O job's global job is that job whose token is cataloged under the name RQGLOBAL in the object directory of the I/O job. For example, if the RQGLOBAL entry in Job A's object directory is a token for Job B, Job B's object directory is the global directory.

   If you wish to share connections among tasks in several jobs, designate one global job. Then catalog tokens for shared connections in the global job object directory.

3. The object directory of the root job. The root job is the first job created when your system is started, All other jobs in the system are offspring of the root job. The root job object directory is available to every job in the system.

   If you wish to share certain connections with all tasks in the system, catalog tokens for the connections in the root job's directory.

CAUTION

Before an I/O job exits, it must uncatalog any objects it cataloged in other directories (global or root). If, for example, a job catalogs the token for a connection in the root job object directory and then exits without uncataloging the token, the logical name and token remain even though the connection is deleted. From then on, using the connection or referring to the logical name will cause an error.

3.7 I/O JOBS

Any job using Extended I/O System calls must be an I/O job. I/O jobs can be created both when the system is initialized, and when programs are running. During configuration, you define the characteristics of I/O jobs that are created when the system is initialized. You use either the CREATE$IO$JOB or RQE$CREATE$IO$JOB system call to create jobs while the system is running. Both of these system calls perform the same operations; the CREATE$IO$JOB system call is compatible with iRMX I systems and reserves up to 1M byte for the jobs memory pool, and the RQE$CREATE$IO$JOB system call gives you the ability to reserve as much as 16M bytes for the job's memory pool. Both system calls are described in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

An I/O job (as opposed to a non-I/O job) must have

- A global job. A token for the global job must be cataloged in the I/O job's object directory under the name RQGLOBAL, as explained in the previous section.
- A default prefix. The default prefix is a connection cataloged under the name $ in either the local job object directory or the global job object directory. Default prefixes are discussed in the next section of this chapter.

- A default user object. This user object must be cataloged in the I/O job's object directory under the name R?IOUSER. A default user object is required to access named files via Extended I/O System calls. Named files and user objects are thoroughly discussed in Chapter 4 of this manual.

RQE$CREATE$IO$JOB automatically initializes the new job with a default user object, global job, and default prefix. ("Automatically" means that these characteristics of a new I/O job are not specified with parameters in the RQE$CREATE$IO$JOB system call, but are inherited from the parent job. Other characteristics such as priority and stack size can be explicitly specified as parameters in the RQE$CREATE$IO$JOB call.)

Any task that invokes the RQE$CREATE$IO$JOB system call must be running within an I/O job. This restriction leads to an obvious question. How can you create the first I/O job? You can create it during the configuration of your application system.

While configuring your application system (described in Chapter 7), you can specify the characteristics of one or more I/O jobs that are created when the system is initialized.

3.8 PATH$PTR PARAMETERS AND DEFAULT PREFIXES

Some Extended I/O System calls refer to files rather than to connections. All such calls require a path$ptr parameter to identify the file to be attached, created, or otherwise manipulated.

The complete interpretation of the path$ptr parameter depends upon the kind of file (named, physical, stream, or remote) being manipulated. Details about this parameter are discussed in Chapter 4 (for named and remote files), in Chapter 5 (for physical files), and in Chapter 6 (for stream files).

However, one aspect of the path$ptr parameter applies to all four kinds of files. If the parameter is set to NIL, or if it points to a null STRING (an iRMX II STRING containing zero characters), the Extended I/O System manipulates the file indicated by the default prefix of the calling task's job.

The default prefix is an attribute of an I/O job, and it is a connection (either a device connection or a file connection). It is cataloged under the name $ in either the local or the global object directory for the job. Whenever a task invokes a system call but does not specify a logical name, the Extended I/O System looks up the default prefix and uses the associated connection.

CHAPTER 4 NAMED FILES

4.1 OVERVIEW

Named files are intended for use with random-access, secondary-storage devices such as disk drives, diskette drives, and bubble memories. Named files provide several features that are not provided by physical or stream files. These features include

- Multiple Files on a Single Volume
- Hierarchical Naming of Files
- Access Control

These features combine to make named files extremely useful in systems supporting more than one application and in applications that require more than one file.

Named files can also reside on remote systems. You access remote named files in the same way you access local named files. To access remote files, you must be using the iRMX Networking Software, which is available separately from the iRMX II Operating System.

4.2 MULTIPLE FILES ON A SINGLE VOLUME

As shown in Figure 4-1, your application can use named files to implement more than one file on a single volume. This can be very useful in applications requiring more than one operator, such as transaction-processing systems.

4.3 HIERARCHICAL NAMING OF FILES

The named files feature allows your application to organize its files into a number of tree-like structures, as depicted in Figure 4-1. Each such structure, called a file tree, must be contained on a single volume, and no two file trees can share a volume. In other words, if a volume contains any named files, the volume contains exactly one file tree.

Each file tree consists of two categories of files--data files and directories. Data files (shown as triangles in Figure 4-1) contain the information that your application manipulates, such as inventories, accounts payable, transactions, text, source code, or object code. In contrast, directory files (shown as rectangles) contain only pointers to other files. The purpose of directory files is to give you flexibility in organizing your file structure.

To illustrate this flexibility, take a close look at Figure 4-1. It shows how named files can be useful in multi-user systems. The figure is based on a collection of hypothetical engineers who work for three departments (Departments 1, 2 and 3). Each engineer is responsible for his own files.

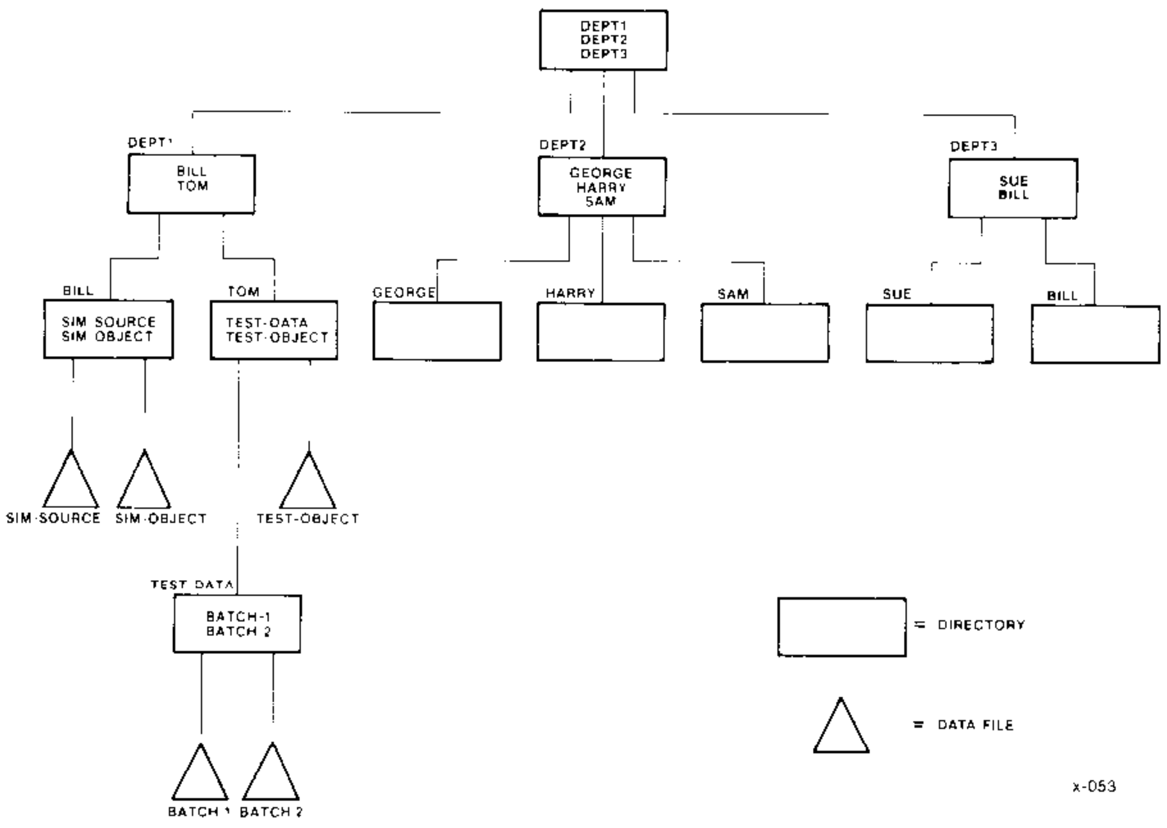


Figure 4-1. Example of a Named File Tree

This multiperson organization is reflected in the file tree. The uppermost directory (called the volume's root directory) points to three "department directories." Each department directory points to several "engineer's directories." The engineers can organize their files as they wish by using their own directories.

The root directory of a remote device is referred to as a virtual root. This is because the remote system selects the directories and files to be made accessible over the OpenNET network. Not all files and directories on a remote system are automatically accessible.

Each file (directory or data) has a unique shortest path connecting it to the root directory of the volume. For instance, in Figure 4-1, the file called SIM-SOURCE has the path DEPT1/BILL/SIM-SOURCE, where the slash (/) is used to separate the components of the path. This notion of "path" reflects the hierarchical nature of the named-file tree.

Another characteristic of hierarchical file naming is that there is less chance for duplicate file names. For example, note that Figure 4-1 contains directories for two individuals named Bill. (These directories are on the extreme left and right of the third level of the figure.)

Even if the rightmost Bill had a data file with the file name of SIM-OBJECT, its path would differ from that leftmost Bill's SIM-OBJECT. Specifically, the leftmost SIM-OBJECT is identified by

DEPT1/BILL/SIM-OBJECT

whereas the rightmost SIM-OBJECT would be identified by

DEPT3/BILL/SIM-OBJECT

Now that you know what a named file is, let's look at how the tasks of your application tell the Extended I/O System which named file to manipulate.

4.3.1 System Calls Requiring Connections

Once you have a file connection for a particular named file, you can use a token for the connection as the connection parameter in any of the following system calls to perform I/O through the connection:

S$CLOSE
S$DELETE$CONNECTION
S$GET$CONNECTION$STATUS
S$OPEN
$READ$MOVE
S$SEEK
S$SPECIAL
S$TRUNCATE
S$WRITE$MOVE

However, if the connection was created by a task in a different job, your task should not use the connection in any of these system calls. Rather, your task should first obtain a new connection to the same file by performing the following steps:

1. Catalog the current connection in the object directory of your task's job. This establishes a logical name for the current connection.
2. Using the newly-defined logical name, invoke the S$ATTACH$FILE system call to obtain another connection to the same file. Your task can use this second connection to invoke any of the system calls listed above.

If your task does attempt to use a connection created in another job, the Extended I/O System will return an exception code rather than performing the requested function.

4.3.2 System Calls Requiring Paths

To use any of the following system calls, your tasks must use an Extended I/O System path, rather than a connection, to tell the Extended I/O System which file you wish to manipulate:

S$ATTACH$FILE
S$CHANGE$ACCESS
S$CREATE$DIRECTORY
S$CREATE$FILE
S$DELETE$FILE
S$GET$DIRECTORY$ENTRY
S$GET$FILE$STATUS
S$GET$PATH$COMPONENT
S$RENAME$FILE

For named files, an Extended I/O System path has two components. The first component is called a prefix, and the second is called the subpath. Let's examine these components one at a time.

4.3.2.1 Prefixes

A prefix is a logical name for a connection to either a device, a named directory file, or a named data file. The device may be either a local or remote device. The files may also be either local or remote files. The purpose of the prefix is to tell the Extended I/O System where to begin interpreting the subpath. The prefix is the only component that is used to distinguish a local connection from a remote connection. Let's look at each of the possible interpretations that the Extended I/O System can derive from a prefix:

- If the prefix is a connection to a local device, the Extended I/O System begins scanning the subpath at the root directory of the device.
- If the prefix is a connection to a remote device, the Extended I/O System begins scanning the subpath at the virtual root directory of the device.
- If the prefix is a connection to a local or remote named directory file, the Extended I/O System begins scanning the subpath at the specified directory.
- If the prefix is a connection to a local or remote named data file, the Extended I/O System checks to see if the subpath is null. If it is, the Extended I/O System manipulates the file indicated by the prefix. If the subpath is not null, the Extended I/O System returns an exception code indicating that your application program is attempting to use a data file as though it were a directory file.

All other syntax applies to both local and remote files. For more information on remote files, see the *iRMX Networking Software User's Guide.*

4.3.2.2 Subpaths

A subpath is a data-file name or a sequence of directory names optionally followed by a data-file name. For instance, referring to Figure 4-1, TOM/TEST-DATA/BATCH-1 is a subpath that leads from the DEPT1 directory to the data file named BATCH-1.

Another example from the same figure is TOM, which is a subpath that leads from the directory named DEPT1 to the directory named TOM.

4.3.2.3 Using Prefixes in Conjunction With Subpaths

The tasks of your application system can use a prefix in conjunction with a subpath to create a complete path for a named file. The prefix generally refers to a directory, and the subpath generally refers to a directory or data file that is a descendant of the directory indicated by the prefix.

4.3.2.4 Specifying Paths in System Calls

Those system calls that require paths have a path$ptr parameter. The tasks of your application system can use this path$ptr parameter, along with the default prefix, to specify the file to be manipulated.

4.3.2.5 Path Syntax

When your application tasks invoke a system call that requires a path, the tasks must provide a path$ptr parameter. When dealing with named files, this parameter is a POINTER to a STRING (see Appendix A for a definition of STRING) that must be in one of the following four forms:

- NULL STRING

  If the STRING is zero characters long, the Extended I/O System will act on the file indicated by the default prefix of the calling task's job.

- LOGICAL NAME ONLY

  If the STRING consists only of a logical name enclosed in colons, the Extended I/O System will look up the logical name and obtain the associated connection. Then, because the subpath is empty, the Extended I/O System will act on the data file or directory file indicated by the connection.

- SUBPATH ONLY

  The STRING can consist of a subpath without a prefix. The Extended I/O System interprets such subpaths by starting at the directory indicated by the default prefix of the calling task's job. Then the Extended I/O System follows the subpath from directory to directory until it reaches the final component of the subpath. This final component is the file on which the Extended I/O System acts.

  Be aware that whenever the STRING contains a subpath without a logical name, the default prefix must be a logical name for a connection to a device or to a named directory file. If, instead, the default prefix represents a connection to a named data file, the Extended I/O System returns an exception code indicating that your task is attempting to use a data file as a directory.

  The following subpath is an example of the most common form:

  ```
  A/B/C/D
  ```

  where A, B, and C are the names of directory files, and D is the name of either a directory or data file. This example causes the Extended I/O System to start at the default directory and descend to Directories A, B, and C in order. Then it acts on D.

  An example of a less common form of subpath is

  ```
  ↑A/B/C/D
  ```

  where the up-arrow (↑) or circumflex (^) tells the Extended I/O System to ascend one level in the hierarchy of files. In other words, the Extended I/O System would read this example as: "Start with the directory indicated by the default prefix and ascend to its parent. Then descend to directories A, B, and C in order. Then act on File D."

  The Extended I/O System can also accept consecutive up-arrows. For example

  ```
  ^^A/B/C
  ```

  would cause the Extended I/O System to start with the directory indicated by the default prefix and ascend two levels before interpreting the remainder of the subpath.

  Another possibility is for the subpath to begin with a slash (/). For example

  ```
  /A/B/C
  ```

  Whenever the Extended I/O System detects a slash at the beginning of a subpath, the Extended I/O System will start interpreting the remainder of the subpath at the root directory of the device indicated by the prefix.

- LOGICAL NAME FOLLOWED BY SUBPATH

  Your application code can use a STRING consisting of a logical name (enclosed in colons) followed immediately by a subpath. For example

  :F0:A/B/C/D

  The Extended I/O System interprets this example as follows. First, it looks up the logical name F0 in the object directory of the local job, or if necessary, the global or root job. Then it follows the subpath from the directory associated with the connection. So in the example, the Extended I/O System would find the directory associated with F0, and it would step through Directories A, B, and C. Finally, the Extended I/O System would act on File D.

4.4 CONTROLLING ACCESS TO FILES

In most environments where files are shared among multiple users, it is necessary to have a means of controlling which users have access to which files. And among users who have access to a given file, it is frequently necessary to grant different kinds of access to different users. The iRMX II Operating System provides this control by identifying users with user IDs and by embedding access rights for these IDs into the files. This section describes the user ID and file access mechanisms.

4.4.1 Users and User Objects

The iRMX II Operating System uses the concept of "user" to correlate file access to people or to jobs. But the precise definition of "user" depends on the nature of your application.

If your application allows a small group of people to enter information (at terminals, for example), you might want to consider each person (or small group of persons) a user. This allows each individual (or small group) to maintain access different from other individuals (or small groups).

Alternatively, if your application does not interact with people (or allows only one person to interact), you might wish to consider each iRMX II job as a user. This setup would allow your application to control the files that each job can access.

In more general terms, the set of entities that manipulate named files in your system is the set of all users. If you want all of these entities to be able to access any file, you can consider them to be a single user. However, if you want to distribute different access to different collections of these entities, you must divide the entities into subsets, each of which is a separate user.

For example, look at Figure 4-1. As mentioned earlier, all engineers are responsible for their own files. If engineers want to have unique access to their files (perhaps permitting no one else to use their files), each engineer must be a separate user. However, if all engineers are willing to give uniform access to other members of the department, then the department can be a separate user.

4.4.1.1 User IDs

A user ID is a 16-bit number that represents any individual or collection of individuals requiring a separate identity for the purpose of gaining access to files.

4.4.1.2 User Objects

The Extended I/O System uses a special type of object called a user object when determining access rights to files. A user object contains a list of one or more user IDs. Each I/O job has a default user object which defines the access rights for all tasks in that I/O job. When a task in an I/O job attempts to manipulate a file, the Operating System computes access by comparing the user IDs listed in the default user object with information contained in the file itself.

To understand user objects, consider an application in which every person who accesses the system has a separate I/O job and therefore a separate user object. The user object represents the person.

The first ID in the user object is the owner ID. This is the ID of the user whom the object represents. If you think of a user object as a person, the owner ID represents the name of that person. When a person creates files, the Operating System automatically embeds the owner ID of that person's user object into the file, allowing that person automatic access to the file.

The IDs that follow the owner ID represent additional kinds of access that the person has. For example, people often belong to organizations such as athletic clubs and fraternal groups which distribute identity cards to their members. To participate in the organization, people must show their identity cards to prove they are members. The user IDs that follow the owner ID serve the same purpose. They identify the person as one of a select group, all of whom have the same access to a certain set of files.

4.4.1.3 Default User Object for a Job

All I/O operations performed within a single I/O job are performed on behalf of one user object, which is called the default user object. The Extended I/O System assumes that the user object cataloged in the I/O job's object directory under the name R?IOUSER is the default user object for that I/O job.

During the configuration of the Extended I/O System, you set up the default user objects for your initial I/O jobs (the ones that start running immediately upon system initialization). Later, when a task creates an I/O job (via the RQE$CREATE$IO$JOB system call), the new I/O job inherits the default user object of its parent I/O job. That is, the Extended I/O System automatically catalogs the parent job's user object in the new I/O job's object directory under the name R?IOUSER. In this way, default user objects pass from parent jobs to offspring.

To prevent problems, you should consider R?IOUSER to be a reserved name and avoid using it.

4.4.1.4 Supporting Dynamic Logon and iRMX-NET

In a system that supports the dynamic logon facilities of the Human Interface or iRMX-NET, a user definition file (UDF) lists the user name, password (in encrypted form), user ID, and other information about everyone who is allowed to log onto the iRMX II system. The Extended I/O System provides the GET$USER$IDS system call so that you can look up the permitted user ID of any user whose user name you know. This system call is useful for tasks that need to set up user objects based on the information listed in the UDF.

The Extended I/O System also helps control remote file access through the system call VERIFY$USER. This system call validates user names and passwords to ensure file security. As a result, the Extended I/O System allows users who log onto dynamic logon terminals controlled by the Human Interface to access remote files.

4.4.2 Types of Access to Files

Each of the two kinds of named files--directory files and data files--can be accessed in four different ways.

Every directory file can potentially be accessed in one or more of the following ways:

Delete	Delete the directory file with S$DELETE$FILE or rename the directory file with S$RENAME$FILE.
List	Obtain the contents of the directory file with S$READ$MOVE.
Add Entry	Add entries to the directory with S$CREATE$FILE, S$CREATE$DIRECTORY, or S$RENAME$FILE.
Change Entry	Change the access rights of files listed in the directory with S$CHANGE$ACCESS.

Every data file can potentially be accessed in one or more of the following ways:

Delete	Delete the file with S$DELETE$FILE or rename the file with S$RENAME$FILE.

Read	Read the file with S$READ$MOVE.
Append	Add information to the end of the file with S$WRITE$MOVE.
Update	Change information in the file with S$WRITE$MOVE or drop information with S$TRUNCATE$FILE.

A user's access rights to a particular file depend on the access list associated with that file.

Access rights to remote files are slightly different than for named files. For more information on access rights to remote files, see the *iRMX Networking Software User's Guide*.

4.4.3 File Access List

For each named file (data or directory), the Operating System maintains an access list which defines the users who have access and their access rights. Each access list is a collection of up to three ordered pairs having the form

ID, access mask

The ID portion is a user ID. The list of user IDs defines the users who can access the file.

The access mask portion defines the kind of file access that the corresponding user has. An access mask is a byte in which individual bits represent the various kinds of access permitted or denied that user. When such a bit is set to 1, it signifies that the associated kind of access is permitted. When set to 0, the bit signifies that the associated kind of access is denied.

iRMX-NET uses a slightly different access mask for remote files than is used for local files.

The association between the bits of the access mask and the kinds of access they control are as follows (where bit 0 is the least-significant bit):

Bit	Directory Files	Data Files
0	Delete	Delete
1	List	Read
2	Add Entry	Append
3	Change Entry	Update

The remaining bits in the access mask have no significance.

For example, an access list for a data file might look like the following:

```
5B31 00001110
9F2C 00000010
```

where the ID numbers (left column) are in hexadecimal and the access masks (right column) are in binary. This means that the ID number 5B31 has read, append, and update access rights, while the ID number 9F2C has the read access right.

The first entry in the file's access list is placed there automatically by the Extended I/O System when it creates the file. The ID portion of that entry is the first ID number in the default user object of the calling task's I/O job. That ID is known as the owner ID for the file. The Extended I/O System fills out the access rights portion to grant the owner ID full (unlimited) access to the file.

Tasks can alter the access list of a file by means of the S$CHANGE$ACCESS system call. With S$CHANGE$ACCESS, you can add or delete ID-access pairs, and you can change the access rights of IDs already in the access list.

NOTE

The user whose ID is the owner ID for a file has one advantage over other users. Other than the system manager user (described later), only a file's owner can use the S$CHANGE$ACCESS system call to modify the file's access list without being granted explicit permission to do so.

4.4.4 Computing Access for File Connections

Whenever a task calls S$CREATE$DIRECTORY, S$CREATE$FILE, or S$ATTACH$FILE, the Extended I/O System constructs an access mask and binds it to the file connection object returned by the call. This access mask is constant for the life of the connection, even if the access list for the file is subsequently altered. When the connection is used to manipulate the file, the access mask for the connection determines how the file can be accessed. For example, if the computed access rights for a connection to a data file do not include appending or updating, then that connection cannot be used in an invocation of S$WRITE$MOVE.

When a task calls S$CREATE$DIRECTORY or S$CREATE$FILE, the Extended I/O System supplies an access mask that grants full access to the connection. However, when a task calls A$ATTACH$FILE, the Extended I/O System compares the default user object with the file's access list and computes an aggregate mask.

Figure 4-2 illustrates the algorithm that the Extended I/O System uses during a call to S$ATTACH$FILE. As the figure shows, the Operating System compares the IDs in the default user object with the IDs in the file's access list. The access masks corresponding to matching IDs are logically ORed, forming an aggregate mask.

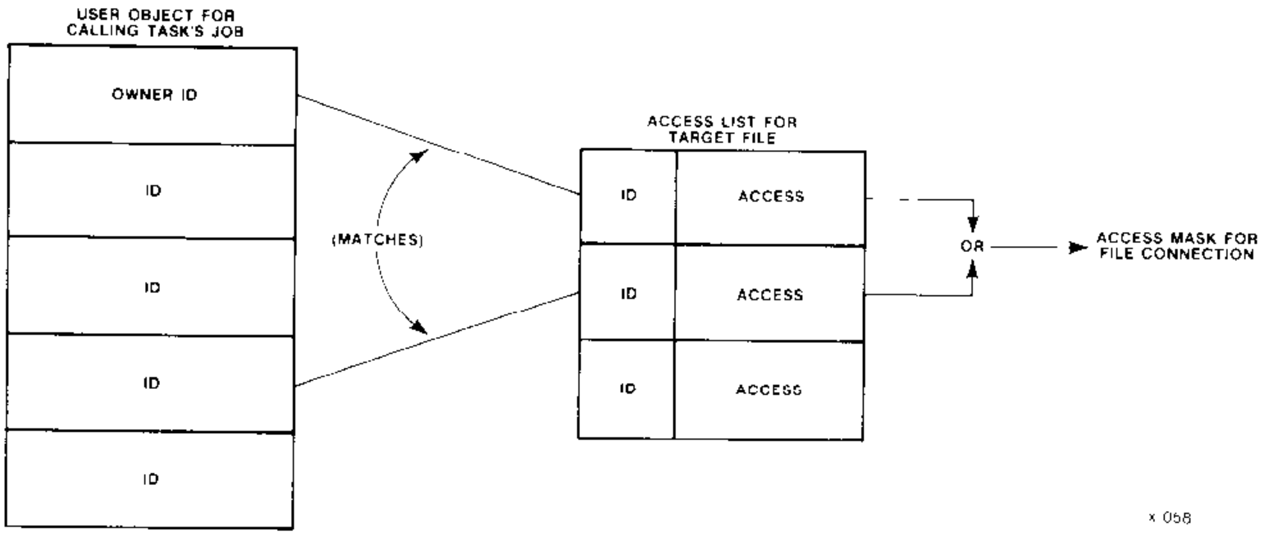

USER OBJECT FOR CALLING TASK'S JOB
OWNER ID
ID
ID
ID
ID
(MATCHES)
ACCESS LIST FOR TARGET FILE
ID
ACCESS
ID
ACCESS
ID
ACCESS
OR
ACCESS MASK FOR FILE CONNECTION
x 058

Figure 4-2. Computing the Access Mask for a File Connection

Normally, the Extended I/O System uses the aggregate access mask embedded in the connection to determine a task's ability to access a file. However, there are two circumstances in which the Extended I/O System computes access again: during S$CHANGE$ACCESS and during S$DELETE$FILE. When a task invokes one of these system calls, the Extended I/O System computes the access to the target file. If the default user object does not have appropriate access rights, the Extended I/O System denies the task the ability to delete the file or change the access.

NOTE

When computing access, the Extended I/O System checks the access only to the last file in the specified pathname and to the parent directory of the last file. It does not check the access to any other directory files specified in the pathname. If the pathname is null, the Extended I/O System checks the access to the file indicated by the default prefix.

4.4.5 Special Users

There are two user IDs that can have special meaning to the Extended I/O System. One is the number 0 (the system manager user) and the other is the number 0FFFFH (the WORLD user).

4.4.5.1 System Manager User

If so indicated during the configuration process, user ID 0 represents the "system manager," or "super user." A user object containing this value is privileged in two respects. First, when it is used to create or attach files, the resulting file connection automatically has read access to data files and list access to directory files. This is true even if a file's access list does not contain an ID-access mask pair whose ID value is 0. The second privilege granted such a user object is that it can call S$CHANGE$ACCESS to change any file's access list.

4.4.5.2 World User

By convention, the user ID 0FFFFH represents WORLD (all users in the system). To implement this convention, you should place the ID for WORLD in the list of user IDs for the initial user objects you set up during the configuration of the Extended I/O System. Then, when your initial I/O jobs create new I/O jobs, the default user objects they inherit will contain the WORLD ID.

By implementing the WORLD convention, your application can set aside certain files as public files, giving everyone limited access. For example, your file system might contain a series of utilities, such as compilers or linkers, which all users need to access. Instead of granting everyone access on an individual basis (which is impossible if you have more than three users), you can grant the user WORLD access to the files. Since WORLD is on the ID list of every user object, this grants everyone access to the files.

As a side effect of including the WORLD ID in every user object, any file whose owner ID is 0FFFFH (WORLD) can have its access list modified by anyone. That is, any file connection for that file can be used in a call to S$CHANGE$ACCESS.

4.5 EXTENDED I/O SYSTEM CALLS FOR NAMED FILES

The Extended I/O System provides a number of system calls that relate to named files. The following sections briefly explain the purpose of each of these system calls. The brief descriptions are grouped by function rather than alphabetically. The *Extended iRMX II Extended I/O System Calls Reference Manual* contains complete descriptions of the Extended I/O System calls.

4.5.1 Obtaining and Deleting Connections

The Extended I/O System provides seven system calls that relate to obtaining and deleting connections.

- S$CREATE$FILE. This call applies only to data files. Your application software must use this call to create a new data file. When an application task invokes this call, the Extended I/O System automatically adds an entry in the parent directory for this new file.
- S$CREATE$DIRECTORY. This call applies only to directory files. When your application software needs to create a directory, the software must use this system call. The call cannot be used to obtain a connection to an existing directory. The Extended I/O System automatically adds an entry in the parent directory for this new directory.
- S$ATTACH$FILE. This call applies to both data and directory files. Your application tasks can use this call to obtain a connection to an existing data file or directory.
- S$DELETE$CONNECTION. This call applies to both data and directory files. Your application tasks can use this call to delete a connection to either kind of named file. This call cannot be used to delete a device connection.
- LOGICAL$ATTACH$DEVICE. This call does not directly apply to either data or directory files. Your application software uses this call to obtain a connection to a local or remote device and to catalog the logical name for the device in the object directory of the root job. Even though this connection is a device connection, it can be used as the prefix (logical name) for the root directory of the device.
- LOGICAL$DETACH$DEVICE. This call does not directly apply to either data or directory files. Your application software uses this call to delete a connection to a device and remove the logical name of the device from the object directory of the root job.
- HYBRID$DETACH$DEVICE. This call is similar to LOGICAL$DETACH$DEVICE in that it deletes a connection to a device. However, HYBRID$DETACH$DEVICE does not remove the device's logical name from the object directory of the root job. Your application software uses this call when it wants to temporarily attach a device in a different manner.

4.5.2 Manipulating Data

Six system calls allow tasks to manipulate the data in a file. All six can be used with data files, while only four apply to directory files. The system calls are

- S$OPEN. This call applies to both data and directory files. Before your application software can use any other system calls to manipulate file data, the software must open a connection to the file. This system call is the only way to open a connection.
- S$CLOSE. This call applies to both data and directory files. After your application software has finished manipulating a file, the application can use this system call to close the file connection. Your application can elect to leave the file open, letting the Extended I/O System close it when the connection is deleted, but closing a file releases memory resources associated with the connection.
- S$SEEK. This system call applies to both data and directory files, and works the same on both file types. Whenever your application software reads, writes, or truncates a file, the requested action takes place at the location specified by the connection's file pointer. The application can tell the Extended I/O System where the operation is to take place. To do this, your application task uses the S$SEEK system call to position the file pointer of the file connection. The S$SEEK system call requires that the file connection be open.
- S$READ$MOVE. This system call applies to both data and directory files. Your application tasks can use this system call to read file data from the location indicated by the file pointer into a segment of memory.

  Before using this system call, your application software can use the S$SEEK system call to position the file pointer. The S$READ$MOVE system call requires that the file connection be open. Also, the segment of memory that receives the information from the file must have write access, because the system call writes the information from the file to memory.
- S$WRITE$MOVE. This system call applies only to data files. Your application software can use this system call to transfer new information from a segment of memory to the file. Before using this call, an application task can use S$SEEK to position the file pointer to the location within the file to receive the information.

  The S$WRITE$MOVE system call requires that the file connection be open. Also, the segment of memory originally containing the information must have read access, because the system call reads the information from memory and writes it to the file.
- S$TRUNCATE$FILE. This system call applies only to data files. Your application software can use this call to trim information from the end of the file. To do so, the application task first can use S$SEEK to position the file pointer to the first byte to be dropped. Then the application invokes the S$TRUNCATE$FILE call to drop all bytes at or beyond the file pointer. The S$TRUNCATE$FILE system call requires that the file connection be open.

4.5.3 Obtaining Status

There are three status-related system calls, one for connections, one for files, and one for devices. The calls are S$GET$FILE$STATUS, S$GET$CONNECTION$STATUS, and GET$LOGICAL$DEVICE$STATUS. The first two calls can be used with data and directory files. The third call retrieves information about devices.

4.5.4 Deleting and Renaming Files

The Extended I/O System provides one system call for deleting files, and another for renaming files. Each of these calls can be used with data and directory files. The calls are

- S$DELETE$FILE. Your application tasks can use this system call to delete data and directory files. However, any attempt to delete a directory that is not empty will result in an exceptional condition.
- S$RENAME$FILE. Your application tasks can use this system call to rename both data and directory files. In renaming a file, an application task can move the file to any directory in the same named file tree. For example, you can rename A/B/C to be A/X/C. In effect, this example simply moves File C from Directory B to Directory X. This means that the application task can change every component of a file's path name except the root directory.

4.5.5 Changing Access

The Extended I/O System provides one system call to let the tasks of your application change a file's access list. This call is S$CHANGE$ACCESS, and it applies to both data files and directories. One rule governs the use of S$CHANGE$ACCESS--only the owner of a file or a user with change entry access to the directory containing the file can change the file's access list.

4.5.6 Deleting Connections

The Extended I/O System provides one system call to delete connections to files. This is the S$DELETE$CONNECTION system call.

4.5.7 Using Logical Names

The Extended I/O System provides three system calls that relate to logical names. All three of these system calls are discussed in detail in the *Extended iRMX II Extented I/O System Calls Reference Manual.*

- S$CATALOG$CONNECTION. This system call allows your application tasks to create a logical name by cataloging a connection in the object directory of a job.

- S$LOOKUP$CONNECTION. This system call accepts a logical name from an application task, looks up the name in the object directories of the local, global, and root jobs (in that sequence), and returns a token for the first connection found. In other words, this is the system call that your application software uses to equate a logical name to a connection.
- S$UNCATALOG$CONNECTION. This system call allows your application software to delete a logical name from the object directory of a job.

4.5.8 Creating and Deleting I/O Jobs

The Extended I/O System provides four system calls that relate to the creation and deletion of I/O jobs.

- CREATE$IO$JOB. This system call creates an I/O job while the system is running. (You can also create one or more I/O jobs when the system is configured, so that they exist when the system starts running.) This system call is available for compatibility with the iRMX I Operating System. If you use this system call to create I/O jobs, the memory pools associated with those I/O jobs cannot exceed 1 megabyte.
- RQE$CREATE$IO$JOB. This system call is just like CREATE$IO$JOB, except that it permits you to assign memory pools of up to 16 megabytes in size. It is recommended that you use this system call (instead of CREATE$IO$JOB) for all new applications, because it takes full advantage of the iRMX II features.
- START$IO$JOB. This call allows you to start the initial task in an I/O job. When you use CREATE$IO$JOB, you specify either that you want the initial task to start running automatically, or that you want to wait until issuing START$IO$JOB.
- EXIT$IO$JOB. This system call provides your application tasks with a convenient method for terminating an I/O job and informing the parent job of the termination.

4.5.9 Performing Miscellaneous Functions

The Extended I/O System provides three system calls to perform miscellaneous operations that do not fit into any other category. The calls are

- S$SPECIAL. Your application tasks can use this system call to perform functions that are peculiar to a particular device. Formatting a disk is an example of such a function. For more information, refer to the S$SPECIAL section of the *Extended iRMX II Extended I/O System Calls Reference Manual.*
- S$GET$DIRECTORY$ENTRY. Your application tasks can use this system call to look up the name of any file in a directory. This is the synchronous version of the A$GET$DIRECTORY$ENTRY system call provided by the Basic I/O System.

- S$GET$PATH$COMPONENT. Your application tasks can use this system call to look up the name of a file as it is known in the file's parent directory. This is the synchronous version of the A$GET$PATH$COMPONENT system call provided by the Basic I/O System.

4.6 BASIC I/O AND NUCLEUS SYSTEM CALLS

Although the purpose of this manual is to describe the Extended I/O System, several system calls provided by the Basic I/O System and by the Nucleus warrant mention here. These calls allow you to specifically manipulate user objects and prefix objects.

The default user and default prefix for each I/O job are cataloged in the job's object directory, and users of the Extended I/O System do not normally manipulate them. Before using the following system calls to alter these objects, refer to Appendix D of this manual.

- SET$DEFAULT$PREFIX (Basic I/O System)
- GET$DEFAULT$PREFIX (Basic I/O System)
- CREATE$USER (Basic I/O System)
- DELETE$USER (Basic I/O System)
- INSPECT$USER (Basic I/O System)
- SET$DEFAULT$USER (Basic I/O System)
- GET$DEFAULT$USER (Basic I/O System)
- CATALOG$OBJECT (Nucleus)
- UNCATALOG$OBJECT (Nucleus)
- LOOKUP$OBJECT (Nucleus)

Refer to the *Extended iRMX II Basic I/O System Calls Reference Manual* and *Extended iRMX II Nucleus System Calls Reference Manual* for complete information on using these calls.

4.7 CHRONOLOGICAL OVERVIEW OF NAMED FILES

Although many system calls can be used with named files, these system calls must be used in a logical sequence.

Figure 4-3 shows the chronological relationships between the most frequently used I/O System calls. To use the figure, start with the leftmost box and follow the arrows. Any path that you can trace is a legitimate sequence of system calls. This figure is not a complete representation of all possible system call sequences.

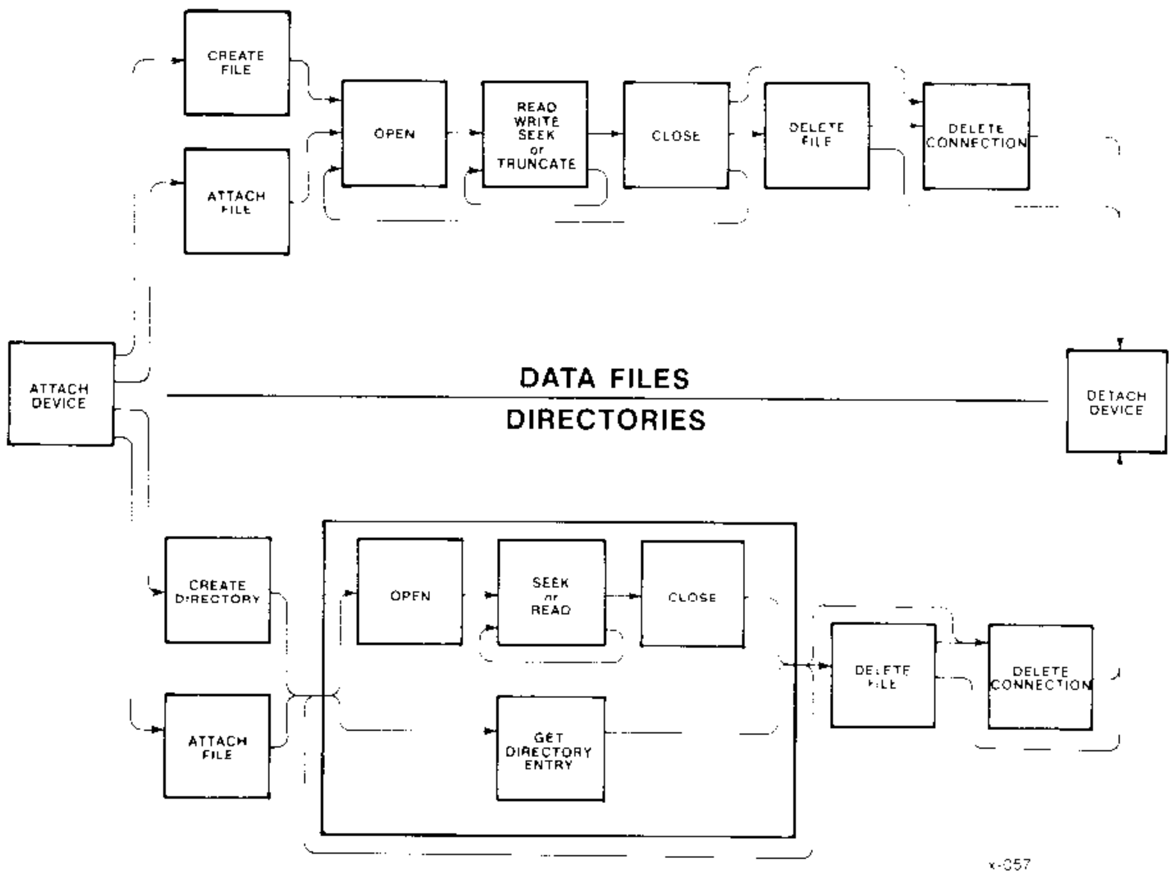


Figure 4-3. Chronology of Frequently Used System Calls for Named Files

CHAPTER 5 PHYSICAL FILES

5.1 OVERVIEW

The iRMX II Extended I/O System provides physical files to allow your applications to read (or write) strings of bytes from (or to) a device. In other words, a physical file occupies an entire device (or the device's entire volume), and the Extended I/O System provides your applications with the ability to access the driver of the device directly.

5.2 SITUATIONS REQUIRING PHYSICAL FILES

The close relationship between a device and a physical file is particularly useful when your application system uses sequential devices. For example, you should use physical files to communicate with line printers, display tubes, plotters, and magnetic tape units.

There are even some instances where you should use physical files to communicate with random devices such as disk drives, diskette drives, and bubble memories. For instance

- Formatting Volumes

  Whenever you create an application task to format a disk or diskette, the task must have access to every byte on the volume. Only physical files provide this kind of access.

- Using Volumes in Formats Required by Other Systems

  If your application tasks must read or write volumes that have been formatted for systems other than the iRMX II Operating System, you must use physical files. Your tasks will have to interpret information such as labels and file structures, but a physical file can provide your tasks with access to the raw information.

- Implementing Your Own File Format

  Suppose that your application system requires a less sophisticated file structure than that provided by iRMX II named files. You can build a custom file structure using a physical file as a foundation.

5.3 CONNECTIONS AND PHYSICAL FILES

Although there is a one-to-one correspondence between the bytes on a device and the bytes of a physical file, the device connection is different from the file connection. The Extended I/O System maintains this distinction to remain consistent with named files and stream files. This consistency helps you develop applications that can use any kind of file.

5.4 SUGGESTION FOR MAINTAINING FILE INDEPENDENCE

If you would like the tasks of your application to be able to use stream or named files in addition to physical files, you should separate the creation of the connection from the use of the connection. For instance, if your application performs I/O to a file, you should consider using two distinct tasks rather than one. The first task would be responsible for obtaining a connection to the file, and the second task would use the connection to perform I/O. By maintaining this separation, you can design the second task to work with any kind of file.

If you choose to use this two-task approach, be sure that both tasks are in the same job. This will eliminate the difficulties associated with passing a file connection from one job to another.

5.5 USING PHYSICAL FILES

Several system calls can be used with physical files, however, the order in which they are used is not arbitrary. The following list provides a brief description (in chronological order) of what an application must do to use a physical file.

1. Obtain a device connection.

   This is necessary for two reasons. When your task creates the physical file, the device connection tells the Extended I/O System which device is to contain the file and that the file must be a physical file.

   You must create a program--here it is called a "system program"--that uses the LOGICAL$ATTACH$DEVICE system call to obtain the device connection. When issuing this call, the system program must use the device name that was assigned to the device during system configuration. For instructions as to how to assign names to devices, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual*.

   Because devices cannot be multiply-attached, you must write your system program so as to call LOGICAL$ATTACH$DEVICE only once. The LOGICAL$ATTACH$DEVICE system call obtains a device connection and catalogs the connection under the logical name provided by the system program. Other tasks wishing to use the device connection can then look up the connection by using the device's logical name.

The LOGICAL$ATTACH$DEVICE system call is described in *Extended iRMX II Extended I/O System Calls Reference Manual.*

2. Obtain a file connection.

   To obtain a file connection, your application task should use one of the following two system calls: S$CREATE$FILE or S$ATTACH$FILE. The decision as to which system call to use depends upon your task's awareness of the existence of the file.

   There are two circumstances under which your task should use S$CREATE$FILE to obtain a connection. The first circumstance is when your task does not know whether the file already exists, and the second is when your task knows that the file does not yet exist.

   When invoking the S$CREATE$FILE system call, set the path$ptr parameter to point to a STRING containing the logical name of the device (enclosed in colons, as in :F0:). This tells the Extended I/O System which device you want as your physical file.

   If, on the other hand, your task is certain that the file already exists, use the S$ATTACH$FILE system call to obtain the file connection. Your task can do this in either of two ways:

   - It can set the path$ptr parameter of the call to point to a STRING containing the device's logical name surrounded by colons (as in :F0:).
   - If the task knows a logical name for a connection to the file, it can set the path$ptr parameter of the call to point to a STRING containing the connection's logical name surrounded by colons (as in :DATABASE:).

   Either way, the Extended I/O System returns a connection to the physical file.

   This careful distinction between the S$CREATE$FILE and the S$ATTACH$FILE system calls is necessary to be consistent with named files. If you want your application to work with named files as well as physical files, you must maintain this consistency.

3. Open the file connection.

   Use the S$OPEN system call to open the connection. When opening the connection, your task must specify whether the task plans to read, write, or do both using the connection. The task must also specify how many buffers the Extended I/O System is to use when reading from or writing to the file. The *Extended iRMX II Extended I/O System Calls Reference Manual* explains how to do this.

4. Manipulate the file.

   There are four system calls that can be used to read, write, or otherwise manipulate your physical file:

   - The S$READ$MOVE and S$WRITE$MOVE system calls are used to read and write information from (to) the physical file.

- The S$SEEK system call can be used to manipulate the file connection's file pointer if the device is a random device such as disk, diskette, or bubble. (If you are writing a device driver for a magnetic tape unit, you can design it to support S$SEEK. Refer to the *Extended iRMX II Device Drivers User's Guide.*)
- The S$SPECIAL system call can be used to request device dependent functions from the device driver. For example, your tasks can use the S$SPECIAL system call to have the Extended I/O System format a disk for use with the iRMX II Operating System. Be aware that use of special functions generally prevent a task from being device independent.

All four of these system calls are described in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

5. Close the file connection.

   Use the S$CLOSE system call to close the connection. Note that your application can repeat steps 2, 3, and 4 any number of times. The S$CLOSE system call is described in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

6. Delete the connection.

   Use the S$DELETE$CONNECTION system call to delete the connection. This is only necessary if the tasks of your application are completely finished using the file. This system call is described in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

7. Request that the device be detached.

   Let the system program know when your task no longer needs the device. The system program can call LOGICAL$DETACH$DEVICE to detach the device. The Operating System keeps track of the number of tasks using the device and avoids detaching the device until it is no longer being used by any task. Only then does the Operating System actually detach the device.

CHAPTER 6 STREAM FILES

6.1 OVERVIEW

Stream files provide a means for one task to send large amounts of information to a second task, even when the two tasks are in different jobs. Be aware that stream files are only one of several techniques for job-to-job communication. If you are not familiar with other techniques, refer to the *Extended iRMX II Programming Techniques Reference Manual.*

The aspect of stream files that makes them very useful is that they allow a task to communicate with a second task as though the second task were a device. This extends the notion of device independence to include tasks.

Since two tasks (called the reading task and the writing task) are involved in using each stream file, the tasks must cooperate. There are a large number of protocols that work, but the ones provided later in this chapter serve as good illustrations.

6.2 SUGGESTION FOR MAINTAINING FILE INDEPENDENCE

If you would like your reading and writing tasks to be able to use named files or physical files rather than only stream files, you should incorporate a third task into the protocol. The purpose of this third task is to perform the one part of the protocol that depends on the kind of file being used--the creation of the file.

6.3 STREAM FILE PROTOCOLS

The interaction between the tasks is divided into three protocols--one each for the creating, writing, and reading tasks. If you choose to avoid using a separate task to create the file, you can have the writing task perform the creating protocol before it performs the writing protocol. However, by eliminating the creating task, you force the writing task to require that only stream files be used. To allow both the reading and writing tasks to be independent of the kind of file being used, you should use a separate creating task.

The following protocols work even if the three tasks are in different jobs. They also work regardless of the order in which they are executed.

6.3.1 Protocol for the Creating Task

The creating task is responsible for obtaining a device connection to the stream file pseudo device, and for creating the stream file. It also must catalog the file connection under a logical name so the reading and writing tasks can attach the file. Remember that this task is not device independent--it works only for stream files. This protocol involves two steps:

1. Creating a stream file.

   While configuring the Extended I/O System, the person that configures your system must enter a configuration parameter that represents a logical name for the stream file device. During system initialization, the Extended I/O System attaches the stream file pseudo device and catalogs the device connection under that logical name. Your tasks can then use the logical name to obtain the device connection.

   To understand this protocol, assume that the logical name is STREAM. (Your system might use a different logical name. To be sure, consult the person(s) responsible for configuring your system.)

   To create a stream file, the creating task need only invoke the S$CREATE$FILE system call using a path$ptr parameter pointing to a STRING of the following form:

   ```
   :STREAM:
   ```

   where STREAM is the logical name for the stream file device connection. The S$CREATE$FILE system call, which is described in the *Extended iRMX II Extended I/O System Calls Reference Manual* returns a connection to the newly created stream file.

2. Catalog the file connection under a logical name.

   The creating task should invoke the S$CATALOG$CONNECTION system call to establish a unique logical name (for example, SF23) for each specific stream file. The reading and writing tasks can then use the logical name to attach the file. The S$CATALOG$CONNECTION system call is described in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

6.3.2 Protocol for the Writing Task

The writing task must perform five steps in order to ensure that it establishes communication with the reading task. The steps are

1. Obtain a connection to the stream file.

   The writing task should use the logical name of the file connection (for example SF23) and invoke the S$ATTACH$FILE system call to obtain the file connection. To do this, the task should set the path$ptr parameter of the system call to point to a STRING containing the file connection's logical name enclosed in colons (as in :SF23:).

2. Open the file connection for writing.

   Use the S$OPEN system call to open the file connection for writing. Set the connection parameter to the token for the file connection, and set the mode parameter to write.

3. Write information to the stream file.

   Use the S$WRITE$MOVE system call as often as desired to write information to the stream file. Use the token for the file connection as the connection parameter.

4. Close the connection.

   When finished writing to the stream file, use the S$CLOSE system call to close the connection. Note that after this step, the writing task can repeat steps 2, 3, and 4 any number of times.

5. Delete the connection.

   Use the S$DELETE$CONNECTION system call to delete the connection to the stream file.

6.3.3 Protocol for the Reading Task

The reading task must perform the following seven steps to successfully read the information written by the writing task:

1. Obtain the file connection for the stream file.

   The reading task should use the file's logical name (for example, SF23) and invoke the S$ATTACH$FILE system call to obtain the file connection. To do this, the task should set the path$ptr parameter of the system call to point to a STRING containing the file connection's logical name enclosed in colons (as in :SF23:).

2. Open the file connection for reading.

   The task should use the S$OPEN system call to open the file connection for reading. Set the connection parameter to the token for the file connection, and set the mode parameter to read.

3. Read information from the stream file.

   The task should use the S$READ$MOVE system call as often as needed to read information from the stream file. Use the token for the file connection as the connection parameter.

4. Close the connection.

   When finished reading from the stream file, the task should use the S$CLOSE system call to close the connection. Note that after this step, the reading task can repeat steps 2, 3, and 4 any number of times.

5. Delete the connection.

   The task should use the S$DELETE$CONNECTION system call to delete the connection to the stream file.

6. Delete the file's logical name created by the creating task.

   The task should use the S$UNCATALOG$CONNECTION system call to delete the logical name for the file. In our example, this logical name is SF23. Do not delete the logical name for the stream file device.

7. Delete the file connection created by the creating task.

   The reading task should use the S$DELETE$CONNECTION system call to delete the file connection that the creating task obtained. Once this connection is deleted, the Extended I/O System automatically deletes the stream file.

CHAPTER 7 CONFIGURATION OF THE EXTENDED I/O SYSTEM

7.1 OVERVIEW

The Extended I/O System is a configurable layer of the Operating System. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, Intel provides three kinds of information:

- A list of configurable options
- Detailed information about the options
- Procedures to allow you to specify your choices

The following sections describe the configurable options. To obtain the second and third categories of information, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

7.2 I/O JOB OBJECTS

When you configure the Extended I/O System, you can specify the following characteristics that deal with iRMX II objects:

- Default size of object directories for I/O jobs and for the Extended I/O System.
- User objects to be created at initialization; each consists of a name for the user object and one or more user IDs. See the section User Objects in Chapter 4 for more information.
- Logical names for devices; these are cataloged in the root job's object directory.

7.3 INTERNAL BUFFER SIZE

You can specify the size of the internal buffers that the Extended I/O System uses to transfer data from files.

7.4 I/O JOB CHARACTERISTICS

When you configure the Extended I/O System, you can create I/O jobs. When the Operating System is initialized, these jobs are also created. (See the *Extended iRMX II Extended I/O System Calls Reference Manual* for a description of I/O jobs.) You can specify as many jobs as you need; the characteristics that you specify for each job correspond to the parameters of the RQE$CREATE$IO$JOB system call. For each job, you specify

- Name of default prefix for this job
- Name of default user object for this job
- Minimum and maximum size of the I/O job's memory pool
- Address of, and mode of, the job's exception handler
- Whether to include parameter validation for the I/O job
- Priority of initial task in job
- Task, data segment, and stack addresses
- Stack size for initial task
- Whether floating-point instructions are used in the job's initial task

7.5 AUTOMATIC BOOT DEVICE RECOGNITION

You can configure the Extended I/O System so that it automatically attaches the device from which the Operating System is bootstrap loaded. This device is then cataloged and know as the system device. Automatic Boot Device Recognition allows you to refer to files on the volume from which the device is bootstrap loaded without knowing the exact physical device on which the files reside. If you want to include this feature, you can specify the characteristics of the System Device.

7.6 INITIALIZATION ERROR REPORTING

During the configuration process, you can elect to have the system report EIOS initialization errors. If you respond "Yes" to the Report Initialization Errors (RIE) parameter on the "Nucleus" screen, the operating system reports initialization errors from all subsystems. On encountering a EIOS initialization error, the operating system returns control to the monitor after writing the following message to the console:

```
   EIOS Initialization Error: <error code number>
```

If Report Initialization Errors is not configured into your system or the iSDM monitor is not present, the initial EIOS task places the EIOS ID code (3) and the corresponding error code into the first two words of the Nucleus data segment (1E0:0000H). It then goes into an infinite error loop.

APPENDIX A DATA TYPES

A.1 DATA TYPES

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

BOOLEAN	A byte that is considered to have a value of TRUE if it is 0FFH, and FALSE if it is 00H. In PL/M 286, DECLARE BOOLEAN LITERALLY 'BYTE';
BYTE	An unsigned eight-bit binary number.
DWORD	A unsigned four-byte binary number.
INTEGER	A signed two-byte binary number. Negative numbers are stored in two's-complement form.
OFFSET	A word whose value represents the distance from the base of an 80286 segment.
POINTER	Two consecutive words containing the selector of a segment and an offset into the segment. The offset must be in the word having the lower address.
SELECTOR	An index into a descriptor table that identifies a particular memory segment. The descriptor table entry lists the segment's base, limit, type, and privilege level.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes that follow it in the string.
TOKEN	A selector that contains the logical address of an object. The selector refers to an entry in the descriptor table that lists the physical address of the object. A token must be declared literally a SELECTOR.
WORD	An unsigned two-byte binary number.

APPENDIX B OBJECT TYPES AND RESOURCE REQUIREMENTS

B.1 OVERVIEW

This appendix lists the type codes for all iRMX II objects. In addition, it documents the resource requirements of the Extended I/O System.

B.2 OBJECT TYPES

Each iRMX II object type is known within iRMX II systems by means of a numeric code. Table B-1 lists the types with their codes.

Table B-1. Type Codes

Object Type	Numeric Code
Job	1
Task	2
Mailbox	3
Semaphore	4
Region	5
Segment	6
Extension	7
Composite	8
User	100
Connection	101
I/O Job	300
Logical Device	301
User-Created Composite	varies from 8000H to 0FFFFH depending on the value specified in CREATE$EXTENSION

The first eight objects, plus user-created composites, are described in the *Extended iRMX II Nucleus User's Guide*. I/O jobs and logical devices are described in Chapter 3 of this manual. Users and connections are described in the *Extended iRMX II Basic I/O System User's Guide*.

B.3 RAM REQUIREMENTS

The following information helps estimate the amount of RAM needed to use the Extended I/O System. The descriptions that follow state explicitly from which pool the RAM is taken. You should use this information when deciding how large to make the memory pools of the jobs in your application. Be aware that this information applies only to the current release of the iRMX II Operating System and may shrink or grow in future releases.

B.3.1 Attaching a Logical Device

Each time one of your tasks uses the LOGICAL$ATTACH$DEVICE system call, the Extended I/O System uses 98 (decimal) bytes of RAM from your job's pool and 64 (decimal) bytes of RAM from the pool of the Extended I/O System job created during the configuration process. This RAM is in addition to the RAM required by the Basic I/O System for a device connection.

Both quantities of RAM are eventually returned to the memory pools from which they originated, but they are returned at different times. The memory taken from the Extended I/O System pool is returned only when the device is detached. In contrast, the memory taken from your job's pool is returned as soon the LOGICAL$ATTACH$DEVICE system call finishes running.

B.3.2 Creating an I/O Job

Whenever one of your tasks creates an I/O job, the Extended I/O System uses 176 (decimal) bytes of RAM from the pool of the I/O job being created. This RAM is in addition to the RAM used by the Nucleus to create the job. All of this memory returns to the pool of the parent job after the I/O job has been deleted.

In addition to the memory requirement, CREATE$IO$JOB and RQE$CREATE$IO$JOB also require five entries in the object directory of the I/O job being created. Refer to Appendix D to see how these entries are used.

B.3.3 Opening a Connection

Whenever one of your tasks uses the S$OPEN system call to open a file connection, the Extended I/O System uses some RAM from the pool of the calling job to create objects. The precise amount of RAM required depends on whether the connection is opened for buffered I/O or nonbuffered I/O. If the connection is not buffered, the Extended I/O System uses 64 (decimal) bytes of RAM. On the other hand, if the connection is buffered, you must use the following expression to compute the amount of RAM used as a function of the buffer size in bytes (S) and the number of buffers (N):

$$\text{number of bytes} = 80 + 5N + N(S+64)$$

Regardless of whether the connection is buffered, all RAM returns to the memory pool when the connection is closed or deleted.

B.3.4 Other RAM Requirements

For system calls other than those discussed above, the Extended I/O System has varying memory requirements. However, you can safely assume that when you make an Extended I/O System call, the call requires no more than

- 300 (decimal) bytes of your job's memory pool.
- 400 (decimal) bytes of the calling task's stack.

This RAM returns to your job's pool as soon as each system call finishes running.

B.4 OBJECT COUNTS

Because each job has a maximum number of objects that it can own, you should know how many objects the Extended I/O System creates while executing system calls. You can assume that the Extended I/O System creates no more than 10 (decimal) objects during the execution of any system call.

Furthermore, except in a few cases, all of these objects are deleted before the system call has finished running. The few exceptions are the system calls that explicitly create objects at the request of your application tasks. Two examples of system calls that explicitly create objects are the S$ATTACH$FILE system call (which creates a file connection) and the LOGICAL$ATTACH$DEVICE system call (which creates a device connection).

APPENDIX C CONDITION CODES

C.1 OVERVIEW

The iRMX II Extended I/O System uses condition codes to inform your tasks of any problems that occur during the execution of a system call. If no problems occur and the system call runs to completion, the Extended I/O System returns an E$OK condition code. Otherwise, the Extended I/O System returns an exceptional condition code.

The meaning of a specific exceptional condition code depends upon the system call that returns the code. For this reason, this appendix does not list any interpretations.

This appendix provides you with the numeric value associated with each condition code that the Extended I/O System can return. To use the exception code values in a symbolic manner, you can assign (using the PL/M-286 "LITERALLY" statement) a meaningful name to each of the codes, or you can use the ERROR.LIT file contained in the :SD:/RMX286/INC directory.

The following list correlates the name of the condition code to the value that the Extended I/O System actually returns. The list is divided into three parts; one for the normal condition code, one for exception codes that indicate a programming error, and one for exception codes that indicate an environmental condition. A programmer error is a condition that is preventable by the calling task. An environmental condition is an exception condition caused by circumstances beyond the control of the calling task. Condition codes are described in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

C.2 NORMAL CONDITION CODE

NAME OF CONDITION	DECIMAL	HEXADECIMAL
E$OK	0	0H

C.3 PROGRAMMING ERRORS

NAME OF CONDITION	DECIMAL	HEXADECIMAL
E$ZERO$DIVIDE	32768	8000H
E$OVERFLOW	32769	8001H
E$TYPE	32770	8002H
E$PARAM	32772	8004H
E$BAD$CALL	32773	8005H
E$NOUSER	32801	8021H
E$NOPREFIX	32802	8022H
E$BAD$BUFF	32803	8023H
E$NOT$LOG$NAME	32832	8040H
E$NOT$DEVICE	32833	8041H
E$NOT$CONNECTION	32834	8042H

C.4 ENVIRONMENTAL CONDITIONS

NAME OF CONDITION	DECIMAL	HEXADECIMAL
E$TIME	1	1H
E$MEM	2	2H
E$LIMIT	4	4H
E$CONTEXT	5	5H
E$EXIST	6	6H
E$NOT$CONFIGURED	8	8H
E$FEXIST	32	20H
E$FNEXIST	33	21H
E$DEVFD	34	22H
E$SUPPORT	35	23H
E$EMPTY$ENTRY	36	24H
E$DIR$END	37	25H
E$FACCESS	38	26H
E$FTYPE	39	27H
E$SHARE	40	28H
E$SPACE	41	29H
E$IDDR	42	2AH
E$IO	43	2BH
E$FLUSHING	44	2CH
E$ILLVOL	45	2DH
E$DEV$OFFLINE	46	2EH

C.4 ENVIRONMENTAL CONDITIONS (continued)

NAME OF CONDITION	DECIMAL	HEXADECIMAL
E$IFDR	47	2FH
E$FRAGMENTATION	48	30H
E$DIR$NOT$EMPTY	49	31H
E$NOT$FILE$CONN	50	32H
E$NOT$DEVICE	51	33H
E$CONN$NOT$OPEN	52	34H
E$CONN$OPEN	53	35H
E$BUFFERED$CONN	54	36H
E$OUTSTANDING$CONNS	55	37H
E$ALREADY$ATTACHED	56	38H
E$DEV$DETACHING	57	39H
E$NOT$SAME$DEV	58	3AH
E$ILLOGICAL$RENAME	59	3BH
E$STREAM$SPECIAL	60	3CH
E$INVALID$FNODE	61	3DH
E$PATHNAME$SYNTAX	62	3EH
E$FNODE$LIMIT	63	3FH
E$LOG$NAME$SYNTAX	64	40H
E$CANNOT$CLOSE	65	41H
E$IOMEM	66	42H
E$MEDIA	68	44H
E$LOG$NAME$NEXIST	69	45H
E$NOT$OWNER	70	46H
E$IO$JOB	71	47H
E$UDF$FORMAT	72	48H
E$NAME$NEXIST	73	49H
E$UID$NEXIST	74	4AH
E$PASSWORD$MISMATCH	75	4BH
E$IO$UNCLASS	80	50H
E$IO$SOFT	81	51H
E$IO$HARD	82	52H
E$IO$OPRINT	83	53H
E$IO$WRPROT	84	54H
E$IO$NO$DATA	85	55H
E$IO$MODE	86	56H
E$IO$NO$SPARES	87	57H
E$IO$ALT$ASSIGNED	88	58H

APPENDIX D USE OF OBJECT DIRECTORIES BY THE EXTENDED I/O SYSTEM

D.1 OBJECT DIRECTORIES

The Extended I/O System catalogs entries in the object directory of each I/O job and in the object directory of the system's root job. This appendix provides a list of the names that the Extended I/O System uses. Do not redefine any of the names listed in this appendix.

RQGLOBAL	The Extended I/O System uses this name to identify the global job for each I/O job. Whenever you create an I/O job, the Extended I/O System automatically catalogs the token for the global job in the object directory of the I/O job. If you wish to redefine this name, you may. But doing so might alter the interpretation of any logical names that are cataloged in the object directory of your job's global job.
R?IOJOB	Whenever you create an I/O job, the Extended I/O System catalogs an object under this name in the object directory of the I/O job. Do not redefine this name!
R?MESSAGE	Whenever you create an I/O job, the Extended I/O System catalogs an object under this name in the object directory of the I/O job. Do not redefine this name!
R?IOUSER	Whenever you create an I/O job, the Extended I/O System catalogs an object under this name in the object directory of the I/O job. Do not redefine this name!
$	The Extended I/O System uses this name to catalog the default prefix for each I/O job. If you modify the definition associated with this name by invoking the CATALOG$OBJECT system call, you change the job's default prefix. Furthermore, if you catalog an object other than a device connection or a file connection under this name, the Extended I/O System generates an exceptional condition code whenever you attempt to use the default prefix.

With the exception of RQGLOBAL and $, you should not use the CATALOG$OBJECT system call to modify any of the definitions described here. If you do change any of them, you might cause the Extended I/O System to behave in an unexpected, unpredictable, and undesirable manner.

The Extended I/O System uses object directories for two other purposes:

- Whenever you use the CATALOG$CONNECTION system call to define a logical name for a connection, the Extended I/O System catalogs the connection in the object directory of the job that you specify.
- Whenever you use the LOGICAL$ATTACH$DEVICE system call, the Extended I/O System catalogs the device connection in the object directory of the system's root job.

APPENDIX E COMPATIBILITY BETWEEN THE TWO I/O SYSTEMS

E.1 COMPATIBILITY BETWEEN THE TWO I/O SYSTEMS

Many of the system calls in the Basic I/O System have counterparts in the Extended I/O System. For example, the A$CREATE$FILE system call of the Basic I/O System performs a function analogous to the S$CREATE$FILE system call of the Extended I/O System. So it is reasonable to ask if connections created by one system can be used by the other.

The answer is yes, unless the connection is open. For example, your application system can use the S$CREATE$FILE Extended I/O System call to create a file and obtain a connection to the file. Because the connection is not open, your application system can use the connection with any Basic I/O System call that does not require an open connection. For instance, the connection can be used with A$RENAME$FILE or with A$GET$FILE$STATUS because neither of these system calls require that the connection be open. However, the connection cannot be used with A$READ or A$WRITE, because both of these system calls require that the connection be open.

The same restriction applies if the connection is created using the Basic I/O System. The connection can be used with any Extended I/O System call as long as the system call does not require an open connection.

In general, you can create, delete, check status, or attach using either kind of system call. But once you have opened the connection, you must use a read, write, truncate, or special-function system call provided by the I/O System that you used to open the connection. Then, once you have closed the connection, you can again use system calls from either I/O System.

APPENDIX F iRMX® I AND iRMX® II EXTENDED I/O SYSTEM DIFFERENCES

F.1 INTRODUCTION

The iRMX II Operating System is based on the iRMX I Operating System. Therefore, the iRMX II version of the Extended I/O System operates almost exactly like its iRMX I counterpart. However, there are a few differences between the two.

The following appendix outlines the differences between the two Extended I/O Systems. These sections are intended for readers who are already familiar with iRMX I and who would like a quick overview of the differences. Those who aren't familiar with either Extended I/O System should skip this appendix.

F.2 EXTENDED MEMORY POOL SIZES

One of the major features of the iRMX II Operating System is its support of the 16M-byte memory-addressing capability of the 80286 processor. The Extended I/O System takes advantage of this feature by allowing the I/O jobs you create to have memory pools as large as 16M bytes. A new system call, RQE$CREATE$IO$JOB, provides this ability.

RQE$CREATE$IO$JOB works exactly like the CREATE$IO$JOB system call available with iRMX I systems. However, two of its parameters (pool$min and pool$max) have been expanded from WORDs to DWORDs. This allows your memory pools to be as large as 16M bytes.

RQE$CREATE$IO$JOB is not a replacement for CREATE$IO$JOB, but an additional system call that has been added to the iRMX II Extended I/O System. CREATE$IO$JOB is still available for compatibility with iRMX I systems. However, if you use CREATE$IO$JOB, your memory pools will be limited to 1M byte.

F.3 PROTECTION FEATURES

The iRMX II Extended I/O System, like the iRMX II Basic I/O System, gives you increased protection. If you try to read from or write into memory segments that do not have the proper access type, or if you attempt to read or write past the end of the segment, you will receive a new exception code called E$BAD$BUFF. The system calls S$READ$MOVE and S$WRITE$MOVE can generate this exception code.

When you use S$READ$MOVE, the buffer of memory used to store the data read in from the peripheral device must have write access (you are writing information to this memory segment). Likewise, when you use S$WRITE$MOVE, the buffer of memory containing the data to be written must have read access (you are reading information from the memory segment).

Other Extended I/O System calls make additional protection checks on the parameters you enter. Supplying incorrect parameters can cause E$PARAM exception codes.

F.4 NEW SYSTEM CALLS

The iRMX II Extended I/O System has two system calls that are not present in the iRMX I Extended I/O System: S$GET$DIRECTORY$ENTRY and S$GET$PATH$COMPONENT. S$GET$DIRECTORY$ENTRY lets you retrieve the name of any file in a directory. S$GET$PATH$COMPONENT looks up the name of a file as it is known in its parent directory. Both of these system calls provide synchronous versions of services that are also available with the Basic I/O System.

A

Access mask 4-11
  bit meanings 4-11
  example 4-12
Advantages of buffers 2-5
Applications using only sequential I/O 1-5
Asynchronous system calls 1-2
Attaching a logical device B-2
Automatic boot device recognition 7-2
Automatic buffering of I/O requests 1-3

B

Buffer size considerations 2-6

C

Characteristics of I/O jobs 7-2
Choosing an I/O system 1-1 to 1-6
Compatibility between the BIOS and EIOS systems E-1
Computing access for file connections 4-12
Condition codes (see Appendix C) C-1
  environmental C-2, 3
  normal C-1
  programming codes C-2
Configuring the Extended I/O system, see chapter 71
Connections and physical files 5-2
Connections between tasks and device units 3-2
Controlling access to files 4-8

D

Data types A-1
Default prefix 3-9, 4-19
Default user 4-19
Default user object for a job 4-9
Device connections 3-5
Device controllers and device units 3-1
Dynamic logon facilities 4-10

E

EIOS algorithm for S$ATTACH$FILE system call 4-13
EIOS system calls for named files 4-15
  changing access 4-17
  creating and deleting I/O jobs 4-18
  deleting and renaming files 4-17
  deleting connections 4-17
  manipulating data 4-16
  miscellaneous functions 4-18
  obtaining and deleting connections 4-15
  obtaining status 4-17
  using logical names 4-17
ERROR.LIT file C-1
Extended I/O System features
  16M-Byte memory addressability 2-1
  access control 2-5
  automatic reattachment of devices 2-7
  buffering with overlapped I/O 2-5
  device independence 2-2
  file independence 2-4
  file sharing 2-5
  four file types 2-3
    named files 2-3
    physical files 2-3
    remote files 2-3
    stream files 2-3
  logical names for files and devices 2-7
  protection features 2-2
  separation of file lookup and file open operations 2-4
  support of many kinds of devices 2-2

F

File access list 4-11
File connections 3-5
File pointers 3-6
Files 3-2
Formula for RAM used by buffered connection B-3
Four string forms permitted for the path$ptr parameter 4-6
Fundamental concepts of the EIOS, see chapter 31

G

General requirements for the EIOS B-3
Global job 3-8, 9

H

Hierarchical naming of file 4-2
High performance applications using Random I/O 1-5

I

I/O jobs 3-8
  exiting from 3-8
Internal buffer size 7-1
IRMX-NET 4-10

L

Logical names 3-7
  and object directories 3-7
  non case sensitive 3-7
  syntax for 3-7
Logical device object 3-5

M

Maintaining file independence 5-2, 6-1
Memory requirements of the EIOS system 1-4
Multiple files on a single volume 4-1

O

Object directories 3-7, D-1
  $ D-1
  R?IOJOB D-1
  R?IOUSER D-1
  R?MESSAGE D-1
  RQGLOBAL D-1
Object types, numeric codes B-1
Opening a connection B-3
Overview of named files 4-19

P

Path syntax 4-6
Path$ptr parameter 3-9, 4-6
Prefixes
  definition 4-5
Protocols for stream files 6-1
  the creating task 6-2
  the reading task 6-3
  the writing task 6-2

R

R?IOUSER 4-9
RAM needed for the EIOS B-2
Reason for having two I/O systems 1-1
Root job 3-8

S

S$CHANGE$ACCESS system call 4-12
Special user IDs 4-14
  system manager 4-14
  WORLD 4-14
Steps for obtaining a new file connection 4-4
Steps in using physical files 5-2
Stream files, see chapter 61
Subpaths 4-6
Synchronous system calls 1-3
System calls requiring connections 4-4
System calls requiring paths 4-5
System device 7-2
System initialization error reporting 7-3

T

Types of access to directories
  add entry 4-10
  change entry 4-10
  delete 4-10
  list 4-10
Types of access to files 4-10
  append 4-11
  delete 4-10
  read 4-11
  update 4-11
Typical number of objects created by the EIOS during system call execution B-3

U

User IDs 4-9
User objects 4-9
Users and user objects 4-8
Using one buffer with the EIOS system 2-6
Using physical files 5-2
Using prefixes and subpaths together 4-6
Using random access devices as physical files 5-1
  formatting volumes 5-1
  implementing your own file format 5-1
  using volumes formatted for other systems 5-1
Using two or more buffers with the EIOS system 2-6

V

VERIFY$USER system call 4-10
Volumes 3-1

intel®

EXTENDED iRMX® II HUMAN INTERFACE USER'S GUIDE

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual documents the Human Interface, one of the layers of the Extended iRMX II Operating System. It is intended for programmers who wish to write application programs that can be loaded and executed via keyboard commands. This manual is divided into the following chapters:

Chapters 1 and 2	Overview of the Human Interface and discussion of the command line interpreter.
Chapters 3 through 6	Discussions of the four general categories of Human Interface system calls and how to use them when writing commands.
Chapter 7	Description of the necessary elements of a Human Interface command, as well as the required compilation and bind sequences.
Chapter 8	Description of the configurable options of the Human Interface.
Appendixes A and B	Listings of type definitions and string table format.
Appendix C	Listing of the differences between the iRMX II Release 2 and the iRMX 86 Release 7 Human Interface.

This manual does not describe the commands supplied with the Human Interface. For information about those commands, refer to the *Operator's Guide To The Extended iRMX II Human Interface.*

CONVENTIONS

This manual is intended for the person who designs and implements the commands (the programmer), not for the person who invokes the commands at the terminal. Whenever this manual describes how certain Human Interface features affect the person that invokes the commands, it refers to that person explicitly as the operator.

This manual uses the following notational conventions to illustrate syntax:

UPPERCASE	In examples of system call syntax, uppercase information must be typed as shown. The programmer can, however, enter this information in uppercase or lowercase.
lowercase	In examples of system call syntax, lowercase fields indicate information to be supplied by the programmer. The programmer must enter the appropriate value or symbol for lowercase fields.

< >	Angle brackets surround variable fields in messages displayed by the Operating System. This information can vary from message to message.

All numbers, unless otherwise noted, are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

RELATED PUBLICATIONS

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

- iRMX 286 Networking Software User's Guide, Order Number: 122323
- iAPX 286 Utilities User's Guide, Order Number: 121934

intel® CONTENTS

CHAPTER 1 OVERVIEW PAGE

1.1 Overview Of The iRMX II Human Interface 1-1
1.2 Resident Human Interface Commands 1-2
1.3 Human Interface System Calls 1-2
1.4 Logon Facility 1-3
1.4.1 Resident User 1-3
1.4.2 Nonresident User 1-3
1.4.2.1 Static Logon Terminals 1-4
1.4.2.2 Dynamic Logon Terminals 1-4
1.4.2.3 Network Access 1-4
1.4.2.4 Logging Off 1-5
1.5 Standard Initial Program 1-5
1.6 Multi-Access Support 1-5
1.7 Wild-Card Pathnames 1-6

CHAPTER 2 THE COMMAND LINE INTERPRETER PAGE

2.1 Introduction 2-1
2.2 CLI Features 2-1
2.3 Initialization 2-2
2.4 Command Invocation 2-3
2.5 User Extensions 2-4
2.5.1 Creating User Extensions 2-4
2.5.1.1 Initialization 2-4
2.5.1.2 Processing 2-4
2.5.1.3 Epilogue 2-5
2.5.1.4 Error Handling 2-5
2.5.1.5 A Sample User Extension 2-5
2.5.2 Binding A User Extension 2-10
2.6 Customized Initial Program 2-11

CHAPTER 3 COMMAND PARSING PAGE

3.1 Overview 3-1
3.2 Standard Command Line Structure 3-1
3.3 Parsing The Command Line 3-5
3.4 Parsing Input And Output Pathnames 3-6
3.5 Wild-Card Characters In Input And Output Pathnames 3-10
3.6 Parsing Other Parameters 3-12
3.7 Parsing Nonstandard Command Lines 3-15

3.7.1 Variations On The Standard Command Line 3-16
3.7.2 Other Nonstandard Command Lines 3-17
3.8 Switching To Another Parsing Buffer 3-18
3.9 Obtaining The Command Name 3-20

CHAPTER 4 PAGE
I/O PROCESSING
4.1 Overview 4-1
4.2 Establishing Input And Output Connections 4-1
4.2.1 Using C$GET$INPUT$CONNECTION 4-1
4.2.2 Using C$GET$OUTPUT$CONNECTION 4-2
4.2.3 Example Program Scenario 4-2
4.3 Communicating With The Operator's Terminal 4-3
4.4 Formatting Messages Based On Exception Codes 4-4

CHAPTER 5 PAGE
COMMAND PROCESSING
5.1 Overview 5-1
5.2 Creating A Command Connection 5-1
5.3 Sending Command Lines To The Command Connection And Invoking The Command 5-2
5.4 Priority Considerations 5-3
5.5 Deleting The Command Connection 5-3
5.6 Example 5-3

CHAPTER 6 PAGE
PROGRAM CONTROL
6.1 Overview 6-1
6.2 How The Default Control-C Mechanism Works 6-1
6.3 Providing Your Own Control-C Mechanism 6-2
6.4 A Sample Control-C Task 6-4

CHAPTER 7 PAGE
CREATING HUMAN INTERFACE COMMANDS
7.1 Overview 7-1
7.2 Elements Of A Human Interface Command 7-1
7.2.1 Parsing The Command Line 7-1
7.2.2 Avoiding The Use of Certain System Calls 7-2
7.2.3 Terminating The Command 7-3
7.2.4 Include Files 7-3
7.3 Producing An Executable Command 7-4

CHAPTER 8 PAGE
CONFIGURATION OF THE HUMAN INTERFACE
8.1 Overview 8-1
8.2 Resident Configuration 8-1
8.3 Nonresident Configuration 8-2
8.4 Initialization Error Reporting 8-3

APPENDIX A **PAGE**
HUMAN INTERFACE TYPE DEFINITIONS
A.1 Type Definitions A-1

APPENDIX B **PAGE**
STRING TABLE FORMAT
B.1 String Format B-1

FIGURES

Figure 2-1. User Extension Example (Continued) 2-6
Figure 3-1. C$GETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME Example 3-8
Figure 3-2. C$GET$PARAMETER Example 3-14
Figure 5-1. Command Connection Examples5-4
Figure 6-1. A CONTROL-C Task Example 6-4
Figure B-1. String Table Format B-1

CHAPTER 1 OVERVIEW

1.1 OVERVIEW OF THE iRMX® II HUMAN INTERFACE

The Extended iRMX II Human Interface is a layer of the Operating System that allows console operators to execute commands on two levels: the standard command line interpreter level (CLI) and the Human Interface level (HI). When the Human Interface begins running, it does the following.

- Initiates a logon process that validates operators. On some terminals, this logon process is invisible, because the terminal has been permanently associated with a single operator. On other terminals (those that can be used by many different operators), the logon process involves typing a logon name and a password before gaining access to the system. If an error occurs during initialization, the Human Interface generates an HI initialization error.
- Creates an iRMX II job for each operator logged into the Human Interface. This job (also called the interactive job) furnishes the application environment with commands to execute.
- Assigns an area of memory for the operator (this occurs when the interactive job is created). Any commands that the operator runs use this area of memory. If there is not enough memory in the system to initialize an operator, the system assigns whatever memory is available at the time and issues a warning message to the terminal.
- Starts an initial program (this also occurs when the interactive job is created). The initial program is the operator's interface to the Operating System. It is usually a command line interpreter (CLI), a program that reads its instructions from the terminal. The Human Interface supplies a standard initial program which reads commands from the terminal and executes the commands based on that terminal input. These commands can either be CLI commands (such as ALIAS and BACKGROUND) which are executed in the operator's interactive job or HI commands (such as COPY, FORMAT, and PASSWORD) which run as offspring jobs of the operator's interactive job. You can also supply your own initial program. In fact, there can be a separate initial program for each user.

When an operator enters information at a terminal, the operator communicates with the initial program. With the standard initial program, the operator can invoke a CLI command by entering the command name (optionally specifying parameters), or a HI command by specifying the pathname of the file that contains the command (optionally specifying parameters). The initial program reads the information from the terminal and executes the commands. If a HI command has been entered, the CLI invokes the Human Interface system calls to load the command into main memory from secondary storage, create an iRMX II job for the command (as an offspring of the operator's interactive job), and begin command execution. If a CLI command has been entered, the CLI interprets and executes the command within the CLI.

The Human Interface provides several features that aid both operators and programmers. These features include:

- A set of Intel-supplied commands.
- A group of system calls to aid programmers in writing their own commands.
- A logon facility to validate operators.
- A standard command line interpreter (CLI) with its own set of commands such as ALIAS, HISTORY, and SET.
- Multi-access support.
- Support for wild-card pathnames.

1.2 RESIDENT HUMAN INTERFACE COMMANDS

In addition to the resident Human Interface, Intel supplies a variety of commands which can be used with any application system that includes the Human Interface. Included are:

- CLI commands (such as ALIAS, HISTORY, SUBMIT, SUPER, and others)
- File management commands (such as COPY, DELETE, BACKUP, RESTORE, and others)
- Device and volume management commands (such as ATTACHDEVICE, FORMAT, DISKVERIFY, and others)
- General Utility commands (such as DEBUG, DATE, and others)

The *Operator's Guide to the Extended iRMX II Human Interface* contains complete descriptions of all commands supplied with the Human Interface.

1.3 HUMAN INTERFACE SYSTEM CALLS

The Human Interface provides a set of system calls that programmers can use in commands they write. The following categories of system calls are available:

- Command-parsing system calls
- I/O and message processing system calls
- Command-processing system calls
- Program control system call

The command parsing system calls provide the ability to parse the command line, allowing you to isolate and identify the parameters in a command line. They also allow you to determine the command name and parse other buffers of text. Chapter 3 provides further discussion of the command parsing system calls.

The I/O and message processing system calls allow you to establish connections to input and output files, communicate with the terminal and format exception codes into a ready-to-display form. Chapter 4 provides a further discussion of the I/O and message processing system calls.

The command processing system calls allow you to invoke interactive Human Interface commands programmatically. Chapter 5 provides a further discussion of the command processing system calls.

The program control system call allows you to override the default Control-C handling task provided by the Human Interface. Chapter 6 describes this in further detail.

1.4 LOGON FACILITY

Logon is the process that the Human Interface uses to validate terminal operators. An operator's view of the logon process differs depending on how the terminal is configured. Terminal users can be configured in one of two ways: resident (or recovery resident) user or nonresident user.

1.4.1 Resident User

The resident user or recovery resident user (who gains control only if an initialization error occurs in the configuration files) is defined during system configuration. All of its attributes are defined in the Human Interface memory during the configuration process and are loaded with the system. A resident user does not use any of the system configuration files.

1.4.2 Nonresident User

A nonresident user and its terminal must be defined in iRMX II system configuration files prior to system execution. Nonresident user terminals can be configured in one of two ways: static logon terminals or dynamic logon terminals.

1.4.2.1 Static Logon Terminals

Each static logon terminal is configured to service a specific operator who can be either a resident or nonresident user. If the static terminal has a nonresident user, its attributes are taken from the configuration files, :CONFIG:TERMINALS, :CONFIG:UDF, and :CONFIG:USER/username during logon (see Chapter 4 in the *Guide To The Extended iRMX II Interactive Configuration Utility* for more information). Therefore, when the Human Interface starts running, it has information about the operator such as user ID, the amount of memory available to this operator, and the priority. This means that the logon process is automatic and invisible to the operator. The only way to change the Human Interface's assumptions about static logon terminals is to change the Operating System's user configuration files and restart the Operating System.

1.4.2.2 Dynamic Logon Terminals

Dynamic logon terminals are configured to service many different operators on a request-by-request basis. To determine which operator wants access to the Operating System via a dynamic logon terminal, the Human Interface requests information before allowing the operator to access the system. This information consists of a logon name and a password. The Human Interface verifies that the information entered is valid by checking user configuration files set up by the system manager. Then it sets up the terminal based on the information listed in those files.

Unlike static terminals, dynamic logon terminals have dynamic memory partitions. That is, the Human Interface does not assign any memory to the terminal at system startup. Instead, it assigns the memory when an operator logs on. The amount assigned varies depending on the operator's requirements (as listed in one of the configuration files). The advantage of dynamic-partition terminals is that the memory available to operators varies depending on the needs of the operator.

When the operator logs off a dynamic logon terminal, the memory goes back into the general free space pool. However, if there is no free space left in the system, an operator won't be able to logon.

The *Guide To The Extended irmx II Interactive Configuration Utility* describes how to set up static and dynamic logon terminals with static and dynamic memory partitions, respectively.

1.4.2.3 Network Access

If the iRMX II system is set up as a workstation on an iRMX-NET communications network, any operator who logs onto the system on a dynamic logon terminal automatically becomes a verified user of the network and can access remote files via the iRMX-NET network. Refer to the *iRMX Network Software User's Guide* for more information about the iRMX NET environment.

1.4.2.4 Logging Off

When operators of dynamic logon terminals finish accessing the Operating System, they can use the LOGOFF command to terminate their sessions. Other operators can then log onto the same terminals.

1.5 STANDARD INITIAL PROGRAM

Once an operator logs onto the Human Interface, the Human Interface assigns an initial program to the operator. This initial program is the first program to run. The identity of this initial program is determined by a privileged operator (normally called the system manager) when adding new users to the system. This process is described in the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

Although the initial program can be almost anything - from an editor to a Basic interpreter - the Human Interface supplies a standard initial program called the Human Interface Command Line Interpreter (CLI). If you have used Release 1 of the iRMX II Operating System (which is still available), note that the CLI supplied with Release 2 has been enhanced to include many new features. The function of the Human Interface CLI is to read input from the terminal, allowing the operator to edit that input if necessary, and execute commands (either CLI or HI) based on the input. The CLI provides a number of additional features such as aliasing, background processing, and recalling of previously entered command lines. Chapter 2 discusses the standard CLI in further detail.

1.6 MULTI-ACCESS SUPPORT

The Basic I/O System supports multiple terminals by providing device drivers that communicate with multiple-terminal hardware. The Human Interface adds to this support by providing identification and protection of users based on logon names and user IDs. This support is called multi-access support.

With multi-access support, multiple operators can communicate with the Operating System. At logon, the Human Interface associates each operator with an identification called a user ID, and assigns each operator a separate area of memory in which to run commands. When an operator creates files or attaches devices, the Human Interface marks the operator as the owner of those files or devices. Access to the files by other users depends on the permission granted those users by the owner. The multi-access Human Interface also provides the operator the capability to execute commands, run development programs (like editors, compilers, and so on), and run other application programs.

To run a multi-access Human Interface, the system manager must first set up the proper directory structure and provide several files containing information about the operators that can access the system. However, you can still tailor your system to meet your individual needs by selecting, for each operator, the initial program that runs when that operator accesses the Human Interface. You can choose the standard CLI (supplied with the Human Interface) or a customized initial program. The user description files maintained by the system manager identify this choice to the Human Interface. This process is described in the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

Programmers who write commands do not have to write their code differently for a multi-access Human Interface than for a single-access Human Interface. The only difference a command might experience in a multi-access environment that it wouldn't experience in a single-access environment involves accessing files and devices. When a command is invoked by an operator, the command inherits the operator's user ID. Thus, the command can perform operations only on files and devices to which the invoking operator has access. In a multi-access environment, a command might not be able to access all the files or devices it wants to access.

1.7 WILD-CARD PATHNAMES

The Human Interface supports the use of wild-card characters in file names. This gives the operator a shorthand method of specifying several files in a single reference. The wild-card characters supported by the Human Interface are

?	Matches any single character
*	Matches any sequence of characters (including no characters)

The *Operator's Guide To The Extended iRMX II Human Interface* describes how an operator can use wild-card characters when entering commands.

Programmers who write their own Human Interface commands do not have to provide special code to support wild-card pathnames as long as they use the Human Interface system calls C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME to obtain the file names from the command line. The Human Interface contains the mechanism to interpret the wild cards and return the correct file name to the calling command. Refer to Chapter 3 for more information about these system calls.

Intel®

CHAPTER 2 THE COMMAND LINE INTERPRETER

2.1 INTRODUCTION

A Command Line Interpreter provides an interface between the operator's terminal and the Operating System. The Command Line Interpreter is usually the means for executing Human Interface commands, but it can be almost anything from a Basic interpreter to an editor. The Command Line Interpreter is the operator's initial program. The Human Interface supplies a standard initial program called the Human Interface Command Line Interpreter (CLI). When invoked the CLI provides the operator with line-editing and alias facilities, background processing, session history, terminal definition and execution of its own set of commands. The Human Interface can also operate with a user extension, which allows you to add customized features to the standard CLI, or with a customized CLI. This chapter explains the features and use of the standard CLI, explains how to incorporate user extensions, and lists the rules for writing a customized CLI.

2.2 CLI FEATURES

The Release 2 Human Interface CLI provides a number of features that make it a useful tool in a development environment. They are listed here with a brief description.

Line-editing — Allows operator to re-edit input.

Aliasing — Allows the operator to abbreviate commonly used commands and assign parameters to them. For example, the operator may define:

```
ALIAS PLM = :LANG:PLM286
```

Then when the operator enters PLM A.28, the CLI executes

```
:LANG:PLM286 A.28
```

Alias expanding can be repeated up to five times for ease of use. An example of reiterative expansion is:

```
ALIAS PLM=:LANG:PLM286
ALIAS PNL=PLM #0.P28 NOLIST
```

When the operator enters: PNL SOURCE, the CLI executes:

```
:LANG:PLM286 SOURCE.P28 NOLIST
```

Background processing	Allows the operator to run jobs in a background environment while continuing to invoke commands at the terminal. The operator is notified when a background job is started and when it finishes. It is possible to request a list of the active background jobs or cancel a background job.
Session history	Displays the last 40 commands and allows the operator to select lines for re-editing.
I/O redirection	Allows standard input and output to be directed somewhere other than the operator's terminal.
Set	Allows the operator to perform on-line changes to certain CLI attributes such as, the prompt and the background memory pool size.

The implementation of these features is possible with the following set of CLI commands: ALIAS, BACKGROUND, DEALIAS, HISTORY, JOBS, KILL, LOGOFF, SET, SUBMIT, and SUPER. All of the CLI commands are described in detail in the *Operator's Guide To The Extended iRMX II Human Interface.* If the standard CLI satisfies the needs of your application, you can assign it to each operator as an initial program.

2.3 INITIALIZATION

The Human Interface CLI can be invoked during either static or dynamic logon. During initialization, the Human Interface CLI performs the following operations:

- Initializes the CLI environment
- Calls CLI extensions, if necessary
- Displays a sign-on message
- Creates an iRMX II object called a command connection in which it places information received from the terminal. Refer to Chapter 5 for more information about command connections.
- Attaches or creates the operator's :PROG: directory
- Submits the file :PROG:R?LOGON for processing

After this initial processing, the CLI displays the Human Interface default prompt (-) and reads input from the terminal. Input from the terminal can be a CLI command, a Human Interface command, or a user application program that is to be executed.

2.4 COMMAND INVOCATION

The CLI begins executing the command either after a carriage return or an ESCAPE has been entered. However, before execution, the CLI allows the operator to edit the input line or recall previously entered lines. When input is terminated, the CLI performs the following operations:

- reads the command line from the terminal into a CLI buffer
- expands all aliases
- handles any I/O redirection that may be necessary
- passes control to the user extension procedure CLI$user$process, if applicable (see section on user extensions later in this chapter)
- searches for CLI commands (such as ALIAS, HISTORY, SET), or HI commands (such as COPY, DISKVERIFY, FORMAT).

If the CLI encounters a CLI command, it parses the command within the CLI job, executes the action requested, and if necessary, calls the Human Interface system call C$SEND$COMMAND to continue the processing. If the CLI encounters a HI command or any user application program, it places the information it reads into the command connection (using the Human Interface command C$SEND$COMMAND). After receiving a complete command, the command connection handler:

- Removes the command name portion
- Loads the file containing the command
- Passes the parameters to the command.

It may be necessary to continue an HI command because of its length or simply for ease in understanding. In this case, the CLI recognizes the ampersand (&) mark at the end of a command line as a continuation character, and displays a double asterisk (**) on the continuation line.

It is possible for the operator to recall either the complete continuation line or only part of it. A double asterisk appears on the screen to indicate that a continuation line is being recalled. The operator can then edit the relevant section of the line. However, after the section has been edited, the entire command line is executed. For an example, see Chapter 3 of the *Operator's Guide To The Extended iRMX II Human Interface.*

The CLI displays error messages for each command in the event of certain operator errors. For a complete description of the CLI error messages, see the detailed explanation of each CLI command given in the *Operator's Guide To The Extended iRMX II Human Interface.*

2.5 USER EXTENSIONS

The Human Interface CLI can be extended to include customized functions. With this feature, you can create an initial program that takes advantage of the standard CLI features, such as line-editing and aliasing, and still meets your precise needs. The procedures that you add to the standard CLI to be able to parse commands differently or implement your own commands are called CLI user extensions. This section explains how to extend the CLI to include user extensions.

2.5.1 Creating User Extensions

Creating a user extension involves writing three procedures: an initialization procedure, a processing procedure, and an epilogue procedure. These procedures, described in the following sections, can be combined into one module. Intel supplies an empty default module called HCLUSR.P28 (located in :SD:RMX286/HI) which provides you with null instances of the three procedures. The Human Interface CLI has three entry points to the user extensions, one before each procedure.

2.5.1.1 Initialization

When the CLI is initialized it first defines its own alias tables (the memory area where user defined aliases are stored) and data structures. It then calls the user supplied initialization procedure. If you have tables or data structures to add during initialization, they should be part of the initialization procedure. This procedure is called only once during CLI initialization. The CLI enters the user extension by calling:

```
CALL CLI$USER$INIT(except$ptr);
```

You can bind this procedure to the CLI library supplied with the Human Interface. An example of how to do this is given later in this section.

2.5.1.2 Processing

After each command line (entered either from a terminal or in a SUBMIT file), the CLI translates all aliases, and checks again for user extensions. At this point, you can change a command, perform additional functions before execution, or process the command. To access your user extension, the CLI calls:

```
cont$flag = CLI$USER$PROCESS(command$ptr, except$ptr);
```

where:

command$ptr	a pointer to a STRING containing the expanded alias command ready for execution.

cont$flag	a byte indicating whether the CLI should continue executing the command line modified by the user extension, or ignore it and continue to the user extension epilogue procedure.

2.5.1.3 Epilogue

When the CLI has executed a command (HI command, CLI command, or user supplied command), it calls the epilogue procedure. This procedure can handle error conditions or perform any other functions that cannot be performed until the command has been executed. The epilogue procedure is called by:

```
CALL CLI$USER$EPILOG(except$ptr);
```

This procedure can be bound to the Human Interface CLI library as shown in the example given later in this section.

2.5.1.4 Error Handling

Each of the three user extension procedures returns an error code in the exception pointer, except$ptr. If the procedure returns anything other than E$OK, the CLI outputs an error message in addition to the message issued by C$SEND$COMMAND or the CLI command.

The CLI catalogs the error code generated by the last command under the name R?ERROR in the global directory before executing the user epilogue procedure. You can access this value and use it in your application. However, any changes to R?ERROR are not recognized by the CLI. The following code enables you to access the value in R?ERROR.

```
DECLARE error$t   TOKEN,
        error     BASED error$t WORD,
        except    WORD;
error$t = RQ$LOOKUP$OBJECT (SELECTOR$OF(NIL), @(7,'R?ERROR'),0,@except);
```

After execution of this system call, error will contain the error code that the last command sent to R?ERROR.

2.5.1.5 A Sample User Extension

The following example shows how to create a user extension using the three procedures described above. The user extension illustrated here allows you to measure the time required to execute a CLI command, an HI command, or any application program. The code shown here is a straightforward example. Many special cases have been omitted.

```
/***************************************************************************

    TITLE:     iRMX II CLI user extension example : TIMER

    ABSTRACT:
        Allow the user to measure time required to execute a CLI
       command or any application program.

   USAGE:
           <command> -T

           where command is any command line that the iRMX II CLI
           executes.

***************************************************************************/

HCLUSR: DO;

       /* global declarations */
DECLARE
   CR       LITERALLY       'ODH',
   LF       LITERALLY       'OAH',
   TOKEN    LITERALLY       'SELECTOR',
   STRING   LITERALLY       'STRUCTURE(
                                 length      BYTE,
                                 char(1)  BYTE)';

       /* include files */

$include(/rmx286/inc/error.lit)
$include(/rmx286/inc/bios.ext)
$include(/rmx286/inc/eios.ext)

       /* externals */

convert$dw$decimal:  PROCEDURE(destination$p, destination$max,
                   dw$number, length, excep$p ) EXTERNAL;
DECLARE
   destination$p   POINTER,
   destination$max WORD,
   dw$number       DWORD,
   length    WORD,
   excep$p         POINTER;
END convert$dw$decimal;
```

Figure 2-1. User Extension Example (Continued)

```
DECLARE
   BOOLEAN      LITERALLY  'BYTE',
   FALSE        LITERALLY  '0',
   TRUE         LITERALLY  '0FFH',
   user$co$t    TOKEN,
   no$co        BOOLEAN,
   timer        BOOLEAN,
   time         DWORD;


$subtitle('CLI$User$Init')

/***************************************************************************
    TITLE:     CLI$User$Init

    CALLING SEQUENCE:

        CALL CLI$User$Init( excep$p);

       ABSTRACT:      The iRMX II CLI calls this procedure at
                initialization, to allow the user to initialize
                data structures and variables.

       ALGORITHM:
                attach and open a connection to :CO:
                signal (NO$CO flag) if it was opened successfully.

***************************************************************************/
CLI$User$Init:    PROCEDURE(excep$p) REENTRANT  PUBLIC;


   DECLARE
       excep$p POINTER,
       excep   BASED   excep$p WORD;

   excep = E$OK;
   no$co = FALSE;
   user$co$t = rq$s$attach$file( @(4,':CO:'), excep$p);
    IF excep = E$OK THEN
       CALL rq$s$open( user$co$t, 3, 0, excep$p);
    IF excep <> E$OK THEN
       no$co = TRUE;

   RETURN;

END CLI$User$Init;
```

Figure 2-1. User Extension Example (Continued)

```
$subtitle('CLI$User$Process')

/***************************************************************************

    TITLE:    CLI$User$Process

    CALLING SEQUENCE:

        continue = CLI$User$Process(comm$buf$p, excep$p);

   ABSTRACT:   The iRMX II CLI calls this procedure before executing
               a command line, but after line-editing and alias
               replacement, to allow command manipulation.  If the CLI
               is to continue processing, this procedure returns TRUE. If
               FALSE is returned, the CLI skips this command and calls
               the CLI$user$epilog.

    ALGORITHM:
           IF -t or -T is found in the command line THEN
           DO;
               get the system time;
               remove '-t' from the command line
               set timer = TRUE for use by the CLI$user$epilog
           END;

           Tell the CLI to continue processing RETURN (TRUE)
***************************************************************************/

CLI$User$Process:    PROCEDURE(comm$buf$p, excep$p) BYTE REENTRANT PUBLIC;

   DECLARE
       comm$buf$p        POINTER,
       comm$buf          BASED       comm$buf$p  STRING,
       excep$p           POINTER,
       excep             BASED       excep$p     WORD;

   DECLARE
       index        WORD;

   excep = E$OK;
   IF no$co
      THEN RETURN(TRUE);
   time  = 0;
   timer = FALSE;
```

Figure 2-1. User Extension Example (Continued)

```
          /* search for '-t' */
   index = FINDB(@comm$buf.char, '-', comm$buf.length);
   IF index <> 0FFFFH THEN
   DO;
       IF comm$buf.char(index + 1) = 'T' OR
               comm$buf.char(index + 1) = 't' THEN
       DO;
               /* remove '-t' from the command */
       comm$buf.char(index), comm$buf.char(index + 1) = ' ';
       time = rq$get$time(excep$p);
       IF excep = E$OK THEN
           timer = TRUE;
       END;
   END;  /* direct CLI to continue command processing*/
   RETURN(TRUE);

END CLI$User$Process;



$subtitle('CLI$User$Epilog')

/***************************************************************************
    TITLE:     CLI$User$Epilog

    CALLING SEQUENCE:

        CALL CLI$User$Epilog(excep$p);

   ABSTRACT:   The iRMX II CLI calls this procedure after executing a
               command line to perform command epilogue functions.

    ALGORITHM:]

              IF there is no connection to :CO: or no timer needed for
              this command THEN RETURN

       Calculate the time elapsed since CLI$user$process;
       Write the time message to the screen;
***************************************************************************/
CLI$User$Epilog:    PROCEDURE(excep$p) REENTRANT  PUBLIC;

   DECLARE
       excep$p       POINTER,
       excep         BASED excep$p  WORD;
```

Figure 2-1. User Extension Example (Continued)

```
    DECLARE
        actual      WORD,
        time$str        STRUCTURE(
                              length    BYTE,
                              char(14)  BYTE);

    excep = E$OK;
    IF no$co OR NOT timer THEN
        RETURN;

    time$str.length = 0;

            /* calculate the elapsed time */
    time = rq$get$time(excep$p) - time;
    IF excep <> E$OK THEN
        RETURN;
    CALL convert$dw$decimal( @time$str, SIZE(time$str), time,
                     SIZE(time$str) - 2 , excep$p);
    IF excep = E$OK THEN
    DO;
        actual = rq$S$write$move(user$co$t, @('Elapsed Time: '),
                      14, excep$p);
        IF excep = E$OK THEN
        actual = rq$S$write$move( user$co$t, @time$str.char,
                         time$str.length, excep$p);
        IF excep = E$OK THEN
        actual = rq$S$write$move(user$co$t, @(' Seconds',CR,LF),
                        10, excep$p);
    END;
    RETURN;

END CLI$User$epilog;
END HCLUSR;
```

Figure 2-1. User Extension Example

2.5.2 Binding A User Extension

To use your user extension, bind it to the Human Interface CLI library using the procedure shown below. (It is recommended that you combine the three procedures into one module, but this is not necessary.) If you have called the user extension module MYEXT.P28, you can use this example exactly as it is written. Otherwise, replace MYEXT.OBJ with the name of the object module you wish to bind.

```
:LANG:BND286   &
MYEXT.OBJ, &
/RMX286/HI/HCLI.LIB(HCLI), &
/RMX286/HI/HCLI.LIB, &
/RMX286/HI/HI.LIB, &
/RMX286/LIB/RMXIFC.LIB, &
/RMX286/HI/HUTIL.LIB, &
:LANG:PLM286.LIB  &
RENAMESEG(CODE TO CLI CODE,DATA TO HI DATA,HI CODE TO CLI CODE,    &
HI DATA TO CLI DATA)    &
OBJECT(MYCLI) NOLOAD NODEBUG SEGSIZE(STACK(2400H)) &
RC(DM(10000,0FFFFH))
```

where:

MYCLI	is the name you use to invoke this CLI. For a complete explanation of the parameters, see Chapter 7 of this manual.

Binding your extensions as shown above creates a Human Interface CLI with your user extension. This newly created CLI can then be called by its pathname, MYCLI, as a nonresident CLI during the logon process. If you want the default resident CLI to include user extensions, you should specify the pathname of the user extension module during configuration. For more information see the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

2.6 CUSTOMIZED INITIAL PROGRAM

If the standard initial program or the standard initial program as modified by a user extension does not meet your needs, you have the option of providing your own initial program. The initial program may be similar to the Human Interface CLI, or it may be a completely different kind of program. For example, you could write a CLI that allows access to files in selected directories only. This would prevent an operator from accidentally modifying other files. Or if you want a particular operator to use only Basic-language programs, a Basic interpreter might be the initial program for that operator. You can select the initial program for each operator. For example, you may continue using the iRMX II Release 1 CLI as the initial program. To do this, you simply specify your selection in the user description files maintained by the system manager (refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual*).

If you provide your own initial program, the program must obey the following rules:

- It must initialize its own data segment. The Human Interface does not set the DS register for the CLI.
- It must perform input and output via logical names :CI: and :CO:.

- If it requires the ability to run Human Interface commands, it must create an iRMX II object called a command connection (via the C$CREATE$COMMAND$CONNECTION system call). If the initial program does not create a command connection, it (and any other application tasks) cannot use the following Human Interface system calls:

  C$GET$INPUT$PATHNAME
  C$GET$OUTPUT$PATHNAME
  C$GET$INPUT$CONNECTION
  C$GET$OUTPUT$CONNECTION
  C$SEND$CO$RESPONSE
  C$SEND$EO$RESPONSE
  C$SEND$COMMAND
  C$SET$CONTROL$C
  C$DELETE$COMMAND$CONNECTION

- If it doesn't create a command connection but still wishes to use the Human Interface system calls C$GET$PARAMETER, C$GET$CHAR, and C$BACKUP$CHAR, it must first invoke the C$SET$PARSE$BUFFER system call.
- It must invoke the Extended I/O System call EXIT$IO$JOB to terminate processing. It must not use the PL/M-286 or ASM286 RETURN statement for this purpose.

Refer to the *Extended iRMX II Human Interface System Calls Reference Manual* for detailed descriptions of the Human Interface system calls mentioned in this section. Refer to the iRMX *II Extended I/O System Calls Reference Manual* for information about the EXIT$IO$JOB system call.

CHAPTER 3 COMMAND PARSING

3.1 OVERVIEW

Whenever an operator enters a Human Interface command from the terminal, an initial program associated with that operator reads the information and causes the Operating System to invoke the command. When it invokes the command, the Operating System places the parameters into a parsing buffer. One of the first things that the command must do is to read the parsing buffer, break the command line into individual parameters, and determine the correct action to take based on the number and meaning of the parameters.

NOTE

The Human Interface supplied initial program reads a command and determines if it is a CLI or a HI command before executing it. CLI commands are handled differently than HI commands. This chapter deals only with HI command parsing.

The Human Interface provides several system calls to parse command lines that follow a standard structure. It also provides other system calls to process nonstandard formats. This chapter:

- Defines the standard structure of command lines
- Describes the system calls used to parse commands having this structure
- Discusses how to switch from one parsing buffer to another parsing buffer
- Describes system calls you can use to parse nonstandard commands
- Describes a system call that you can use to obtain the command name the operator used when invoking the command

3.2 STANDARD COMMAND-LINE STRUCTURE

The standard structure of a Human Interface command line consists of a number of elements separated by spaces. It is recommended that your commands follow this structure to enable parsing by the Human Interface system calls. However, if you require a different structure, refer to the "Parsing Nonstandard Command Lines" section of this chapter.

The standard structure is as follows (square brackets [] indicate optional portions):

command-name [inpath-list [preposition outpath-list]] [parameters] <cr>

where:

command-name	Pathname of the file containing the command's executable object code. The pathname may consist of a prefix and a subpath. A prefix is a logical name of a directory and is unique if it is not duplicated in one of the directories in the command search sequence defined during configuration. See the *Extended iRMX II Interactive Configuration Utility Reference Manual* for more details on directories, prefixes and logical names.
inpath-list	One or more pathnames, separated by commas, of files that the Human Interface reads as input during command execution. Individual pathnames can contain wild-card characters to signify multiple files. Refer to the *Operator's Guide To The Extended iRMX II Human Interface* for a description of the wild-card characters and their usage. You can use the C$GET$INPUT$PATHNAME system call to process this inpath-list.
preposition	A word that tells the Human Interface how to handle the output. The standard structure supports the following prepositions:

TO The Human Interface writes the output to a new file indicated by the output pathname. If the file already exists, the Human Interface queries the operator as follows:

<pathname>, already exists, OVERWRITE?

If the operator enters a Y or an R (uppercase or lowercase), the Human Interface overwrites any information in the existing file with the new output. (An R tells the Human Interface to continue overwriting existing files without prompting for permission.) Any other character causes the Human Interface to proceed with the next pair of input and output files.

OVER The Human Interface writes the output to the file indicated by the output pathname. It overwrites any information that currently exists in the file.

AFTER The Human Interface appends the output to the end of the file indicated by the output pathname.

You can use the C$GET$OUTPUT$PATHNAME system call to process the preposition.

outpath-list	One or more pathnames, separated by commas, of files that are to receive the output during command execution. The total number of pathnames in this list and the number of wild-cards used depends on the inpath-list. Refer to the *Operator's Guide To The Extended iRMX II Human Interface* for more information. You can use the C$GET$OUTPUT$PATHNAME system call to process the outpath-list.
parameters	Parameters that cause the command to perform additional or extended services during command execution. The standard structure supports parameters with the following formats:
value-list	The parameter consists solely of one or more groups of characters (called values) separated by commas. When the value-list is present in the command line, the command performs the service indicated by the values.
keyword=value-list	A keyword with an associated value (or list of values, separated by commas). The keyword portion identifies the kind of service to perform, and each value supplies further information about the service request.
keyword(value-list)	Alternate form of the previous format.
keyword value-list	A keyword with an associated value (or list of values, separated by commas). Like the previous two formats, the keyword portion identifies the kind of service to perform and each value portion provides more information about the service. However, the keyword must be identified to the command as a preposition (refer to the description of the C$GET$PARAMETER system call in the *Extended iRMX II Human Interface System Calls Reference Manual* for more information). You use the C$GET$PARAMETER system call to process the parameter.
cr	Line terminator character The RETURN (or CARRIAGE RETURN) key and NEW LINE (or LINE FEED) key are both line terminators.

The following examples show how you should enter an Human Interface command using the command structure described above.

```
COPY /rmx286/file1 TO /rmx/file2 <cr>
FORMAT :f0: FILES=300 GRANULARITY=200 BS <cr>
UPCOPY :f1:myfile TO /newdir/outfile Q <cr>
```

For more examples see the *Operator's Guide To The Extended iRMX II Human Interface.*

The Human Interface also supports the following special characters:

continuation character	An ampersand character (&). When an operator includes an ampersand in the command line as the last character before the line terminator, the Human Interface assumes that the command invocation continues on the next line. If the standard Human Interface command line interpreter (or any custom command line interpreter that uses C$SEND$COMMAND to invoke commands) processes the operator's command entry, the ampersand (and the line terminator that follows) are edited out of the parsing buffer. Then the continuation line is read and appended to the parsing buffer. This process continues until the operator enters a line terminated by a carriage return without a continuation character. Therefore, when the command receives control, its parsing buffer contains a single command invocation, without intermediate continuation characters or line terminators.

NOTE

CLI commands such as ALIAS, SUBMIT and SUPER do not recognize continuation characters. However, continuation characters are recognized by all human Interface commands found in :SYSTEM:.

comment character	A semicolon character (;). The Human Interface considers this character and all text that follows it on a line to be a non-executable comment. If the standard Human Interface command line interpreter (or any custom command line interpreter that uses C$SEND$COMMAND to invoke commands) processes the operator's command entry, all comments are edited out of the parsing buffer. Therefore, individual commands do not have to search for and discard comments.

quoting characters

Two single-quote (') or double-quote (") characters remove the semantics of special characters they surround (but you must use the same character for both the beginning and ending quote). If a command line contains quoted characters, the Human Interface system calls that invoke the command and parse the command line do not perform any special functions associated with the surrounded characters. For example, an ampersand surrounded by double quotes is interpreted as a single ampersand and not a continuation character.

The quotes remove the semantics of characters that are special to the Human Interface but not special to other layers of the Operating System. Therefore, quotes do not remove the semantics of characters such as :, /, and /, which are special to the I/O System.

To include the quoting character in the quoted string, the operator must specify the character twice or use the other quoting character. For example:

```
'can''t' or  "can't"
```

causes:

```
     can't
```

to be read in the command line.

3.3 PARSING THE COMMAND LINE

When a Human Interface command begins executing, a parsing buffer associated with the command contains all the parameters that the operator entered when invoking the command (everything except the command-name portion of the invocation line). The Human Interface maintains a pointer for this parsing buffer which initially points to the first parameter. By invoking any of the following Human Interface system calls, the command can read the parameters from the parsing buffer:

C$BACKUP$CHAR
C$GET$INPUT$PATHNAME
C$GET$OUTPUT$PATHNAME
C$GET$PARAMETER
C$GET$CHAR

The system calls C$GET$INPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and C$GET$PARAMETER read an entire parameter and cause the Human Interface to move the pointer to the next parameter. These system calls understand quoting characters, remove the special meaning from quoted characters, and discard the quote characters.

The system calls, C$BACKUP$CHAR and C$GET$CHAR, see the parsing buffer as a string of characters. They do not understand the notion of quoting characters; therefore they do not remove the special meaning from quoted characters, nor do they skip over the quotes. C$BACKUP$CHAR causes the Human Interface to move the pointer one position backwards. C$GET$CHAR reads a single character and causes the Human Interface to move the pointer to the next character. Except for positioning the parsing pointer to a particular place in the buffer, C$GET$CHAR should not be used with C$GET$INPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and C$GET$PARAMETER.

3.4 PARSING INPUT AND OUTPUT PATHNAMES

If you restrict the invocation lines of the commands you write to a form that is similar to the standard format discussed earlier in this chapter, you can use the system calls C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME to identify the input and output pathnames in the command line. Because the command line can contain multiple pathnames, you might need to invoke these system calls several times to obtain all the pathnames.

The first call to C$GET$INPUT$PATHNAME reads the entire inpath-list (the list of pathnames separated by commas) into a buffer, moves the parsing pointer to the next parameter, and returns the first input pathname to the command. Likewise, the first call to C$GET$OUTPUT$PATHNAME notes the preposition (TO, OVER, or AFTER), reads the entire outpath-list into a buffer, moves the parsing pointer to the parameter after the outpath-list, and returns the first output pathname to the command. Succeeding C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME calls return additional pathnames from the buffers created previously, but they do not move the parsing pointer to the next parameter.

For example, if the parsing buffer contains:

```
A,B TO C,D
```

the first call to C$GET$INPUT$PATHNAME obtains both input pathnames (A and B), returns the first one (A) to the caller, and positions the pointer at the preposition TO. The first call to C$GET$OUTPUT$PATHNAME obtains both output pathnames (C and D) and returns the first one (C) to the caller. C$GET$OUTPUT$PATHNAME also identifies TO as the preposition and positions the pointer on it. The second calls to C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME return B and D respectively to the caller.

These system calls handle single pathnames, lists of pathnames, and pathnames containing wild-card characters. However, because of this versatility and because output pathnames are dependent on input pathnames when both use wild-card characters, you must make calls to C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME in a particular order. To use these system calls effectively, obey the following rules:

1. Always call C$GET$INPUT$PATHNAME to obtain the input pathname before calling C$GET$OUTPUT$PATHNAME to obtain the corresponding output pathname. This is necessary because with wild-card characters, the identity of the output pathname depends on the identity of the input pathname. Therefore, C$GET$OUTPUT$PATHNAME cannot determine the output pathname until C$GET$INPUT$PATHNAME determines the corresponding input pathname.
2. Always alternate your calls to C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME. This is necessary to handle wild-card characters and lists of pathnames. If you invoke two calls to C$GET$INPUT$PATHNAME without an intermediate call to C$GET$OUTPUT$PATHNAME, you will not be able to obtain the first output pathname. Similarly, if you invoke two calls to C$GET$OUTPUT$PATHNAME without an intermediate call to C$GET$INPUT$PATHNAME, the second call returns invalid information.

C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME return the pathnames in the form of iRMX II strings. Each string is a group of bytes in which the first byte contains the number of ASCII bytes that follow. For these system calls, the remaining bytes in the string contain the pathname. If C$GET$INPUT$PATHNAME returns a zero-length string (that is, the first byte is zero), you know that there are no more pathnames to obtain.

After calling C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME to obtain the input file and corresponding output file, you can use the system calls C$GET$INPUT$CONNECTION and C$GET$OUTPUT$CONNECTION to obtain connections to those files. Chapter 4 contains more information about C$GET$INPUT$CONNECTION and C$GET$OUTPUT$CONNECTION. Upon obtaining connections to the files, you can perform the necessary I/O operations.

Figure 3-1 contains an example of a program that uses C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME in its command-line parsing (it also uses C$GET$INPUT$CONNECTION and C$GET$OUTPUT$CONNECTION to obtain connections to the files). This program is a partial example of a COPY command that you could implement.

```
/*************************************************************************
*  This example demonstrates the use of the following Human Interface    *
*  system calls:                                                         *
*                                                                        *
*      rq$C$get$input$pathname                                           *
*      rq$C$get$output$pathname                                          *
*      rq$C$get$input$connection                                         *
*      rq$C$get$output$connection                                        *
*                                                                        *
*  This program is a possible implementation of a COPY utility whose     *
*  purpose is to copy data from successive input files to corresponding *
*  output files.  For example, to copy file A to file B, file C to file *
*  D, and file E to file F, an operator could specify the following      *
*  command line:                                                         *
*                                                                        *
*      COPY A,C,E TO B,D,F                                               *
*************************************************************************/

copy: DO;

DECLARE token LITERALLY 'SELECTOR';
DECLARE (input$pathname, output$pathname)  STRUCTURE (
                                           length       BYTE,
                                           char (41)    BYTE ),
       output$prep BYTE,
       (input$token, output$token) TOKEN,
       excep       WORD,
       exit$excep  WORD;

$include (/rmx286/inc/error.lit)
$include (/rmx286/inc/hi.ext)
$include (/rmx286/inc/eios.ext)

/* Get the first input pathname string  */
CALL rq$C$get$input$pathname (@input$pathname, SIZE(input$pathname), @excep);

   IF excep <> E$OK        THEN
   CALL rq$exit$io$job (excep, NIL, @exit$excep);
```

Figure 3-1. C$GETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME Example

```
   DO WHILE (input$pathname.length <> 0);  /* A zero length indicates no
                    more input parameters.*/
   /*  Get the corresponding output pathname string  */
       output$prep = rq$C$get$output$pathname (@output$pathname,
                                              SIZE(output$pathname),
                                             @(7,'TO :CO:'), @excep);
       IF excep <> E$OK        THEN
           CALL rq$exit$io$job (excep, NIL, @exit$excep);

   /*  Establish connection with the pair of input and output files */

       input$token = rq$C$get$input$connection (@input$pathname, @excep);
       IF excep <> E$OK        THEN
           CALL rq$exit$io$job (excep, NIL, @exit$excep);

       output$token = rq$C$get$output$connection (@output$pathname,
                                                output$prep, @excep);
       IF excep <> E$OK        THEN
           CALL rq$exit$io$job (excep, NIL, @exit$excep);


           .
           .       Code to copy data and close both files
           .


   /*  Get the next input pathname string */
       CALL rq$C$get$input$pathname (@input$pathname,
                                     SIZE(input$pathname),@excep);
       IF excep <> E$OK        THEN
           CALL rq$exit$io$job (excep, NIL, @exit$excep);

   END  /* DO WHILE */

/* Finish I/O processing */
CALL rq$exit$io$job (excep, NIL, @exit$excep);

END copy;
```

Figure 3-1. C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME Example

3.5 WILD-CARD CHARACTERS IN INPUT AND OUTPUT PATHNAMES

Wild-card characters provide a shorthand notation for specifying several files in a single reference. The Human Interface supports two wild-card characters for use in the last component (this means the last parameter, not just the last character) of input or output pathnames. The wild-card characters are:

?	The question mark matches any single character. For example, the name "FILE?" could imply all of the following names (and more): FILE1 FILE2 FILEX
*	The asterisk matches any sequence of characters (including zero characters). For example, the name "*FILE" could imply all of the following files (and more): OBJECTFILE FILE V1.2FILE AFILE

The following example illustrates both the correct and incorrect usage of a wild-card as the last component in a list of pathnames. You can enter:

```
/mylib/file*
    or
/mylib/*.obj
```

Entering

```
/*lib/file1
```

will cause an error. The *Operator's Guide To The Extended iRMX II Human Interface* describes how to use wild-card characters when entering commands. It also discusses restrictions and operational characteristics of which an operator should be aware. Refer to that manual for more information about using wild-card characters in file names.

The C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME system calls automatically handle pathnames that contain wild-card characters. They treat a wild-carded pathname as a list of pathnames.

C$GET$INPUT$PATHNAME matches wild cards. That is, each time you call it, it compares the wild-carded component with the files in the specified directory and returns the pathname of the next file that matches. For example, if an input pathname is:

```
:PROG:PLM/A*
```

C$GET$INPUT$PATHNAME searches the :PROG:PLM directory and returns the pathname of the next file that begins with the letter "A".

C$GET$OUTPUT$PATHNAME generates wild cards. Each time you call it, it compares the wild-carded output pathname with the wild-carded input pathname and with the most recent pathname returned by C$GET$INPUT$PATHNAME. Then it generates a corresponding output pathname based on that information. The output pathname could refer to an existing file or to a file that does not yet exist. As an example, suppose an operator's default directory contains the following files:

ALPHA	BETA
A11	B11
ADAM	C11

Now suppose that you have written a command called REFINE that reads some information from an input file, adjusts that information in some manner, and writes the information to an output file. Assuming that you interleaved the calls to C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME correctly when you wrote the command, an operator could enter a command line as follows:

```
REFINE A*,B* TO C*,D*
```

In this case, C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME return pathnames as follows:

Pathname list returned by C$GET$INPUT$PATHNAME	Corresponding pathname list returned by C$GET$OUTPUT$PATHNAME
ALPHA	CLPHA
A11	C11
ADAM	CDAM
BETA	DETA
B11	D11

Because the file C11 already exists, the Human Interface would display the following message before writing to the file:

```
    C11, already exists,  OVERWRITE?
```

If the operator answers yes to the prompt, the Human Interface overwrites the file.

3.6 PARSING OTHER PARAMETERS

The C$GET$PARAMETER system call is also available for parsing command lines of the standard format. You can use this system call for the following purposes:

- To parse parameters which appear after the input and output pathnames.
- To parse all parameters, if the command does not use input and output files.
- To parse the input and output pathnames, if the command requires a preposition other than TO, OVER, or AFTER.

If you use C$GET$PARAMETER to parse input and output pathnames, you must provide additional code to handle wild-card characters that may appear in the command line. This is unlike C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME which handle wild-card characters automatically. For example, suppose a command line contains the pathname:

```
FILE*
```

If you use C$GET$INPUT$PATHNAME to parse this parameter, the system call assumes that FILE* is a wild-carded pathname. It searches the operator's default directory and returns the pathname of the first file whose name starts with the characters "FILE". Subsequent calls to C$GET$INPUT$PATHNAME return other pathnames that meet these conditions.

However, if you use C$GET$PARAMETER to parse the same parameter, the system call returns the value:

```
FILE*
```

It does not know that the characters represent a pathname, nor does it know that the asterisk represents a wild card.

When called, C$GET$PARAMETER parses a single parameter and moves the pointer of the parsing buffer to the next parameter. The parameter returned as a result of this call can be in any of the following forms:

value-list	A value or group of values separated by commas. The system call returns the entire list in the form of a string table (described in Appendix B). It places each of the values in the value list in a separate string.

keyword = value-list or keyword (value-list)	A keyword indicating the kind of parameter, followed by a value (or group of values,separated by commas). The presence of the equal sign or the parentheses lets the system call recognize keyword parameters without foreknowledge of the keywords. It also informs the system call that the characters following the equal sign (or the characters in parenthesis) represent a value-list and not a separate parameter. The system call returns the keyword in a string and the value-list in a string table.
keyword value-list	A keyword indicating the kind of parameter, followed by a value (or group of values, separated by commas). In this case, since the keyword and value-list are separated by spaces instead of by an equal sign or parentheses, the keyword is referred to as a preposition. In order for the system call to recognize that this structure is a keyword/value-list instead of two separate parameters, you must supply, as input to the system call, a string table containing all the possible prepositions that could occur. The system call checks this list to determine whether a group of characters separated by spaces is a preposition keyword or a separate parameter.

Individual parameters are separated by spaces.

In general, the value-list of a parameter is either a single value or a list of values separated by commas. C$GET$PARAMETER returns each of these values as a string in a string table. However, an individual value can itself consist of a value-list. If a group of values (separated by commas) is enclosed in parentheses, C$GET$PARAMETER treats the values as a single value, returning them in a single string. For example, in the following value-list:

```
A,(B,C,D),E
```

C$GET$PARAMETER considers "B,C,D" as a single value. Therefore, the value-list consists of three values: "A", "B,C,D", and "E".

Figure 3-2 contains an example of a program that uses C$GET$PARAMETER in its command-line parsing.

```
/************************************************************************
*  This example demonstrates the use of the following Human Interface    *
*  system call:                                                          *
*                                                                        *
*      rq$C$get$parameter                                                *
*                                                                        *
*  This program makes use of rq$C$get$parameter to parse a keyword       *
*  parameter in a command line.  Here, the keyword, "SIZE", is parsed    *
*  and its value portion converted to a word value and placed in         *
*  "size$val".  For example, an operator could specify the following     *
*  command line:                                                         *
*                                                                        *
*      PROG1  SIZE = 400                                                 *
*                                                                        *
*  Note that if the "SIZE" parameter is not present, "size$val" receives *
*  a default value.                                                      *
************************************************************************/
prog1: DO;

DECLARE token LITERALLY 'SELECTOR';

$include (/rmx286/inc/error.lit)
$include (/rmx286/inc/hi.ext)
$include (/rmx286/inc/eios.ext)

DECLARE STRING                LITERALLY 'STRUCTURE (len BYTE,
                                                     str (1) BYTE)',
       STRING$TABLE          LITERALLY 'STRUCTURE (num$entries BYTE,
                                                 entries (1) BYTE)',
       PARAMETER$KEYWORD$MAX LITERALLY '20',
       VALUE$TABLE$MAX       LITERALLY '80',
       DEFAULT$SIZE          LITERALLY '100';

DECLARE value$table$buf (VALUE$TABLE$MAX) BYTE,  /* Receives string table
                                                     value  */
       value$table  STRING$TABLE AT (@value$table$buf),
       value$str$ptr  POINTER,
       value$str BASED value$str$ptr STRING;   /* For referencing strings
                                                   in the string table  */

DECLARE parameter$keyword$buf (PARAMETER$KEYWORD$MAX) BYTE, /* Receives
                                                   the keyword
                                                   string */
       parameter$keyword STRING AT (@parameter$keyword$buf),
       excep WORD,
       more$param WORD,
       (size$val, i) WORD;
```

Figure 3-2. C$GET$PARAMETER Example

```
/* Get the next parameter, if present */
more$param = rq$C$get$parameter(@parameter$keyword, PARAMETER$KEYWORD$MAX,

                                @value$table, VALUE$TABLE$MAX,
                                NIL,NIL,@excep);
IF (excep = E$OK) AND (more$param) THEN
   IF (parameter$keyword.str(0) = 'S') AND  /* Is the keyword 'SIZE'? */
      (parameter$keyword.str(1) = 'I') THEN
       DO;
         value$str$ptr = @value$table.entries; /* Point to 1st entry in
                                                  table */
         size$val = 0;
         DO i = 0 to value$str.len - 1;  /* Convert number string to word
                                            value */
               size$val = size$val * 10;
               size$val = size$val + (value$str.str(i) - 30H);
         END;
       END;
   ELSE
       size$val = DEFAULT$SIZE; /*If the 'SIZE' parameter is not present,
                                  use the default size. */


                     .
                     .   Continue with the rest of the program
                     .

       /*  Finish I/O processing */
CALL rq$exit$io$job (excep, NIL, @exit$excep);
END prog1;
```

Figure 3-2. C$GET$PARAMETER Example

3.7 PARSING NONSTANDARD COMMAND LINES

If the command line you write follows the recommended structure described earlier in this chapter, you can use C$GET$INPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and C$GET$PARAMETER to parse the command line. However, if you require an invocation line of a different form, you might not be able to use these system calls. The following sections discuss two types of nonstandard command lines: one that is similar to the standard and one that is completely different.

3.7.1 Variations On The Standard Command Line

The "Standard Command-Line Structure" section of this chapter recommends that the first parameters of your commands be a list of input pathnames, a preposition, and a list of output pathnames. With this convention, commands always call C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME first, before obtaining any optional parameters. Therefore, the input and output pathnames are the only position-dependent parameters in your commands; other parameters can appear in any order and can be optional.

However, suppose you want to structure your commands so that other parameters appear before the input and output pathnames. You can still use C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME to parse the input and output pathnames. But, you have to ensure that your command knows which of the parameters contain the input and output pathnames. You can do this in several ways. Two of them are:

- Enforce a rigid structure on the command line. For example, suppose you want two parameters to appear before the input and output pathnames, such as:

  ```
  command p1 p2 input-pathname prep output-pathname
  ```

  Your command could use C$GET$PARAMETER to parse the first and second parameters. Then it could use C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME to parse the input and output pathnames. If you do this, p1 and p2 are position-dependent parameters which must be included whenever the command is invoked.

- Use a separate parameter as a switch to inform the command that the parameters that follow are input and output pathnames. This method requires more code to implement but it can allow you to make all your parameters (including the input and output pathnames) position-independent.

  For example, you could implement your command such that whenever the operator entered a parameter called FROM, it would signal the command that the next parameters were input and output pathnames. This command could contain a main loop that used C$GET$PARAMETER to parse parameters. Whenever the command received a parameter whose value was "FROM", it could call another portion of code that used C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME. After retrieving the input and output pathnames, the code could return to the main loop to continue processing parameters.

  A hypothetical command of this sort might be called RETRIEVE, a command that retrieves information from various data bases. The operator could invoke this command with a command line such as:

  ```
  RETRIEVE NAMES ADDRESSES PHONES FROM file1 TO file2
  ```

  In this command, operators can specify what they want to retrieve before they specify where to get the information.

3.7.2 Other Nonstandard Command Lines

In some instances, you might want your command line to look completely different from that described earlier in this chapter. For example, suppose you require a syntax in which the following rules apply:

- Spaces have no significance and can be omitted between parameters.
- A prefix character must be before each parameter (a "$" indicates an input file, an "@" indicates an output file, and a "-" indicates all other parameters).

With this kind of syntax, a user could invoke a command (in this example the command is called REFINE) as follows:

```
REFINE $infile-medium@outfile
```

where infile is the file from which to read information, outfile is the file in which REFINE should place its output, and medium is a parameter that further directs the processing.

If you require the syntax outlined in this example (or any other nonstandard syntax), you cannot use C$GET$INPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and C$GET$PARAMETER to parse the individual parameters. Any of these system calls would return the entire parameter list as a single parameter.

For cases such as this, you can use the C$BACKUP$CHAR and the C$GET$CHAR system calls to parse the command line. These system calls perform a single, simple operation. C$BACKUP$CHAR moves the position pointer backwards one position. C$GET$CHAR returns a single character from the command line and moves the pointer to the next character. These system calls do not understand the notion of parameters as explained earlier in this chapter. Nor do they understand wild-card characters or quoting characters.

C$BACKUP$CHAR and C$GET$CHAR require you to provide the parsing algorithm in your own program, because they make no assumptions about the structure or order of parameters. However, by using these system calls, you can enforce any command syntax you choose.

Because C$BACKUP$CHAR and C$GET$CHAR move the pointer character by character, not parameter by parameter, you should take care when using them in the same program with C$GET$INPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and C$GET$PARAMETER. You must ensure that C$BACKUP$CHAR and C$GET$CHAR leave the pointer pointing at the beginning of a parameter (or at blank characters which immediately precede the parameter) before invoking any of the other system calls.

3.8 SWITCHING TO ANOTHER PARSING BUFFER

When a command begins execution, it has a parsing buffer that is set up by the Human Interface to contain the parameters of the command. The command parsing system calls listed in this chapter operate on that parsing buffer. This allows the command to parse its parameters.

Some commands might require the ability to parse additional lines of text (for example, an editor needs to parse individual editor commands) after the original command invocation. A command such as this cannot use the Human Interface-provided parsing buffer because it has no way of placing information in the buffer, and because it cannot reset the parsing pointer to the beginning of the buffer.

To meet the needs of commands such as this, the Human Interface provides a system call to change the parsing buffer from the one the Human Interface provides to one that the command provides. This system call, C$SET$PARSE$BUFFER, switches the parsing buffer and sets the parsing pointer to the beginning of the buffer.

One of the parameters of the C$SET$PARSE$BUFFER system call (buff$p) is a pointer to a buffer containing the text to be parsed. This buffer can contain text read from the terminal, text read from a file, or even text that you "hard code" into the command. After the call to C$SET$PARSE$BUFFER, the following command parsing system calls obtain information from the new parsing buffer:

C$GET$PARAMETER
C$GET$CHAR
C$BACKUP$CHAR

The other command parsing calls (C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME) are not affected by calls to C$SET$PARSE$BUFFER. These calls always obtain pathnames from the original parsing buffer (the command line).

When you establish a new parsing buffer, C$SET$PARSE$BUFFER sets the parsing pointer to the beginning of the buffer. This allows you to use one buffer for parsing many lines of text. For example, suppose your command has several sub-commands. Each time the operator enters a sub-command, your command reads the sub-command into a buffer, calls C$SET$PARSE$BUFFER to reset the parsing pointer, and parses the sub-command. The program flow for an operation like this could be:

1. Read the information from the terminal into a buffer (use C$SEND$CO$RESPONSE, C$SEND$EO$RESPONSE, or an Extended I/O System call).
2. Call C$SET$PARSE$BUFFER to set the parsing buffer to the buffer containing the sub-command. This sets the parsing pointer to the beginning of the buffer.

3. Parse the sub-command using C$GET$PARAMETER, C$BACKUP$CHAR or C$GET$CHAR system calls.
4. Perform the operations requested by the sub-command.
5. Go back to step 1. Continue this loop until the operator exits from the command.

If you specify NIL or a zero value for the buff$p parameter of C$SET$PARSE$BUFFER, the parsing buffer switches back to the original command line buffer. However, the parsing pointer does not reset to the beginning of the buffer; it remains pointing at the next parameter in the command line. This allows you, if you wish, to parse part of the command line, switch buffers and parse a portion of another buffer, and switch back to the command line.

There is one problem with switching back and forth between parsing buffers. Except when you switch to the command line buffer, every time you call C$SET$PARSE$BUFFER, the parsing pointer moves to the start of the buffer. Therefore, you lose your place in the buffer. However, C$SET$PARSE$BUFFER returns, in its offset parameter, a value that indicates the position of the pointer in the previous buffer. This value specifies the offset of the pointer, in bytes, from the beginning of the buffer. If you intend to switch back to that buffer (by again calling C$SET$PARSE$BUFFER), you can use this value to move the pointer to its previous position.

One way to do this is to use the C$GET$CHAR system call to move the parsing pointer back to its previous position. After switching back to the original buffer, call C$GET$CHAR the number of times specified in the offset parameter of the first C$SET$PARSE$BUFFER call (not the one that switched back to the buffer). This positions the pointer to its previous location. You can then continue parsing parameters from the point at which you left off.

Another way to do this is by treating your parsing buffer as an array of characters (an array called CHAR, for example). When you call C$SET$PARSE$BUFFER the first time, you can specify the buff$p parameter to point to the first element of the array (CHAR(0), for example). Then, when you switch parsing buffers, C$SET$PARSE$BUFFER returns, in the offset parameter, the number of bytes already parsed. When you switch back to the first parsing buffer, you can use this offset value as an index into the array; that is, have the buff$p parameter point to CHAR(offset).

3.9 OBTAINING THE COMMAND NAME

A user invokes a command by specifying the pathname of the file containing its object code and any parameters the command requires. The Human Interface places the parameters in a parsing buffer, which the command can access by invoking the system calls described earlier in this chapter. In addition, the Human Interface places the command name in another buffer. The command can obtain this name by calling C$GET$COMMAND$NAME.

C$GET$COMMAND$NAME does not operate on the parsing buffer used by the other command parsing system calls. Nor is it affected by the C$SET$PARSE$BUFFER system. It can be called multiple times; each time it returns the same command name.

If the operator enters the complete pathname of the command (including the logical name), the command-name buffer contains exactly what the operator entered. However, if the operator enters a command name without a logical name, the Human Interface automatically searches a number of directories for the command. In this case, the command-name buffer contains not only the name the operator entered, but also the directory containing the command (such as :SYSTEM:, :PROG:, or :$:).

Therefore, a command can use the value returned by C$GET$COMMAND$NAME and the circumflex pathname separator (^) to access the directory in which it resides. For example, if "command-name" is the name received from C$GET$COMMAND$NAME, a command could access its directory by using the pathname:

```
command-name/
```

It could access another file in the directory by specifying the pathname:

```
command-name/file
```

CHAPTER 4
I/O PROCESSING

4.1 OVERVIEW

The Human Interface provides several system calls that establish connections to input and output files, communicate with the operator's terminal, and format exception codes into messages that can be sent to the operator. This chapter discusses these system calls.

4.2 ESTABLISHING INPUT AND OUTPUT CONNECTIONS

The Human Interface provides two system calls for establishing connections to input and output files: C$GET$INPUT$CONNECTION and C$GET$OUTPUT$CONNECTION. These system calls are structured so that you can use the output from C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME as input to these system calls.

4.2.1 Using C$GET$INPUT$CONNECTION

C$GET$INPUT$CONNECTION obtains a connection to a file and opens that connection for reading. One of the parameters of C$GET$INPUT$CONNECTION is a pointer to a string containing the pathname of the file for which the connection is sought. This pathname can be the pathname returned by C$GET$INPUT$PATHNAME or it can be the pathname of any other file to which you want a connection. If C$GET$INPUT$CONNECTION cannot obtain a connection to the specified file for any reason, it returns an exception code and writes a message to :CO: (normally the operator's terminal) to indicate the type of problem. For example, if the specified input file does not exist, C$GET$INPUT$CONNECTION displays the following message:

```
<pathname>, file not found
```

The system call displays similar messages in other situations. Refer to the description of C$GET$INPUT$CONNECTION in the *Extended iRMX II Human Interface System Calls Reference Manual* for more information.

Because C$GET$INPUT$CONNECTION returns messages to the operator in the event of an exceptional condition, your command does not have to return additional messages unless you require them. The command must decide only whether to abort or to continue processing.

4.2.2 Using C$GET$OUTPUT$CONNECTION

C$GET$OUTPUT$CONNECTION obtains a connection to a file and opens that connection for writing. As in the case of C$GET$INPUT$CONNECTION, one of the parameters of C$GET$OUTPUT$CONNECTION is a pointer to a string containing the pathname of the file for which a connection is sought. This pathname can be the pathname returned by C$GET$OUTPUT$PATHNAME or it can be the pathname of any other file to which you want a connection. There is another parameter in C$GET$OUTPUT$CONNECTION which specifies the type of preposition to use when writing to the output file (TO, OVER, or AFTER). This preposition governs how data gets written to the file.

If you specify the TO preposition and the pathname of an existing file, C$GET$OUTPUT$CONNECTION prompts the operator for permission to delete the existing file. This prompt appears as:

```
<pathname>, already exists, OVERWRITE?
```

If the operator enters a "Y" or "y" (for "yes"), the system call obtains the connection to the existing file. If the operator enters an "R" or "r" (for "repeat"), the system call also obtains the connection to the existing file, and it gives the system call permission to obtain additional output connections, if necessary, without prompting for permission to delete existing files. If the operator enters any other character, the system call returns an exception code without obtaining a connection to the file.

If you specify the OVER preposition, C$GET$OUTPUT$CONNECTION obtains the connection without prompting the operator for permission.

If you specify the AFTER preposition, C$GET$OUTPUT$CONNECTION obtains the connection without prompting the operator for permission. It also seeks to the end of file before returning control. Thus, any information you write to the file will not overwrite the existing information. This is unlike TO and OVER which cause C$GET$OUTPUT$CONNECTION to leave the file pointer at the beginning of the file.

If the operator does not have the proper access rights to the file, or if for some reason C$GET$OUTPUT$CONNECTION cannot obtain a connection to the file, C$GET$OUTPUT$CONNECTION returns an exception code and displays a message at the operator's terminal. Refer to the description of C$GET$OUTPUT$CONNECTION in the *Extended iRMX II Human Interface System Calls Reference Manual* for more information.

4.2.3 Example Program Scenario

A normal scenario for using C$GET$INPUT$CONNECTION and C$GET$OUTPUT$CONNECTION is as follows:

```
DO WHILE more input and output files

    Obtain input pathname from command line with C$GET$INPUT$PATHNAME
    Obtain output pathname from command line with
C$GET$OUTPUT$PATHNAME
    Obtain connection to input file with C$GET$INPUT$CONNECTION
    Obtain connection to output file with
C$GET$OUTPUT$CONNECTION
    Read information from input file
    Perform command operations on information
    Write information to output file
    Delete connections to input and output files

END
```

The program listing in Figure 3-1 shows a partial implementation of this scenario.

4.3 COMMUNICATING WITH THE OPERATOR'S TERMINAL

The Human Interface provides two system calls that ease the process of communicating with the operator's terminal. They are C$SEND$CO$RESPONSE and C$SEND$EO$RESPONSE. Each of these system calls combines into a single system call several operations that you would normally perform when communicating with the terminal.

In its general form, C$SEND$CO$RESPONSE establishes connections to :CI: (console input) and :CO: (console output), writes a message to :CO:, and reads a message from :CI:. As input to this system call, you can specify the message to be sent, the size of the message to be received, and the buffer to receive the message. Depending on the values you choose for the parameters, you can either:

- Send a message and receive a message
- Send a message without waiting to receive a message
- Receive a message without sending a message

If you use C$SEND$CO$RESPONSE, you do not have to invoke other system calls to attach, open, read from, or write to the operator's terminal.

There is a difference between C$SEND$CO$RESPONSE and C$SEND$EO$RESPONSE. C$SEND$CO$RESPONSE deals specifically with the logical names :CI: and :CO:. Therefore, its input and output can be redirected to files by changing the pathnames represented by these logical names. This is what happens when an operator places a command in a SUBMIT file; SUBMIT assumes that :CI: is the SUBMIT file and that :CO: is the output file specified in the SUBMIT command. On the other hand, while C$SEND$EO$RESPONSE performs the same operations as C$SEND$CO$RESPONSE, C$SEND$EO$RESPONSE always reads information from and writes information to the operator's terminal. Input and output cannot be redirected with C$SEND$EO$RESPONSE.

C$SEND$EO$RESPONSE is especially useful if you have multiple tasks communicating with a single terminal. If a task uses either of these system calls and requests a response from the terminal, no other output is displayed at the terminal until the operator enters a response to the first system call. After the operator responds, tasks can send further information to the terminal. This mechanism, when used by all the tasks which communicate with the terminal, prevents the operator from receiving several requests for information before being able to respond to the first one.

4.4 FORMATTING MESSAGES BASED ON EXCEPTION CODES

Whenever you include iRMX II system calls in the code of a command that you write, it is possible for those system calls to encounter exceptional conditions. Exceptional conditions are divided into two categories: programming errors and environmental conditions. Programming errors occur when the iRMX II Operating System detects a condition that normally can be avoided by correct coding. Environmental conditions, in contrast, are generally outside the control of the application program.

Even the most thoroughly debugged commands can encounter exceptional conditions. The exceptional conditions can arise from invalid operator entries, lack of secondary storage space, media errors, and other problems over which the command has no control.

The Human Interface provides a default exception handler to handle exceptional conditions in commands that you write. This exception handler receives control on the occurrence of all exceptional conditions. It displays the exception code value and mnemonic at the operator's terminal and aborts the command.

In many cases, you might want to provide your own exception handling, either to pass additional information to the operator or to allow the operator another chance to enter correct information. In such cases, you can use the Nucleus system calls GET$EXCEPTION$HANDLER and SET$EXCEPTION$HANDLER to assign your own exception handler or to cancel the effect of the default exception handler on some or all exceptions that occur in your command. Refer to the *Extended iRMX II Nucleus System Calls Reference Manual* for more information about these system calls.

When you perform your own exception handling, you will probably create special messages that you return to the operator in the event of certain exceptional conditions. However, you might not want to create messages for all possible exception codes. For this situation, the Human Interface provides the C$FORMAT$EXCEPTION system call.

C$FORMAT$EXCEPTION accepts an exception code value as input and returns a string whose contents describe the exceptional condition. You can use this string as input to a system call such as C$SEND$CO$RESPONSE to write the information to the operator's terminal. By using C$FORMAT$EXCEPTION, you can return a message to the operator for all exceptional conditions, but you do not have to enlarge your program by including the text of these messages in the code of your command.

The text portion of the string produced by C$FORMAT$EXCEPTION consists of the exception code value and mnemonic in the following format:

value : mnemonic

You can display this string as is, or you can place additional explanatory text in the string before displaying it. The following example shows you how to use C$FORMAT$EXCEPTION. Suppose you have a procedure named DONOTHING that writes an error message to the screen whenever this procedure encounters an exception. You can declare a message as follows:

```
DECLARE
    error$msg   STRUCTURE(
                        length      BYTE,
                        char(80)    BYTE),
failed(*)       BYTE DATA(31,'DONOTHING procedure failed *** '),
excep          WORD,
local$excep    WORD;
```

Now, whenever an exception is encountered during execution, you can call C$FORMAT$EXCEPTION, as shown below, to create the default message for the exception contained in the excep variable and concatenate it to the failed message you declared in the variable failed:

```
CALL MOVB(@failed, @error$msg, failed(0));

CALL rq$C$format$exception(@error$msg, SIZE(error$msg), excep, 1,
                           @local$excep);
```

You can write the error$msg string to the screen. For example, if the excep variable contains 05H, the string contained in error$msg would be

```
' DONOTHING procedure failed *** 0005: E$CONTEXT'
```

Refer to the *Extended iRMX II Human Interface System Calls Reference Manual* for more information about C$FORMAT$EXCEPTION.

intel®

CHAPTER 5 COMMAND PROCESSING

5.1 OVERVIEW

When you write your own command, you might want to perform an operation that is already provided in another command (such as copying one file to another, displaying a directory, etc.). Instead of duplicating the code for this operation in your command, you can invoke Human Interface system calls to issue the commands themselves. The effect of making these system calls is the same as that produced by an operator entering a Human Interface command at the terminal. The Human Interface provides three system calls to facilitate this process of programmatic command invocation: C$CREATE$COMMAND$CONNECTION, C$SEND$COMMAND, and C$DELETE$COMMAND$CONNECTION.

Invoking commands programmatically involves the following operations:

- Creating an object (called a command connection) to store the command invocation lines
- Sending the command line to the command connection and invoking the command
- Deleting the command connection

This chapter discusses these operations and provides an example of how the Extended iRMX II system calls appear in a program.

5.2 CREATING A COMMAND CONNECTION

Before you can send a command line to the Operating System to be invoked, you must create an object (called a command connection) to store the command line. The C$CREATE$COMMAND$CONNECTION system call creates this object and returns a token for the command connection. The token can be used in calls to C$SEND$COMMAND (to send command lines to the object) and in calls to C$DELETE$COMMAND$CONNECTION (to delete the object after using it).

When you call C$CREATE$COMMAND$CONNECTION, you also specify tokens for the connections that serve as command input and command output for the invoked command. This allows you to redirect input and output for the invoked command to secondary storage files. Or you can specify the normal :CI: and :CO:.

The command connection is necessary to support the processing of multiple-line commands without interference from other tasks. If not for the command connections, the Operating System would be unable to determine which continuation line went with which command when many tasks were sending command lines to be processed. The command connection provides a place to store command lines until the command is complete.

5.3 SENDING COMMAND LINES TO THE COMMAND CONNECTION AND INVOKING THE COMMAND

The C$SEND$COMMAND system call sends command lines to a command connection and, when the command invocation is complete, invokes the command. One of the parameters of this system call is the token for a command connection, which identifies the command connection to use. Another parameter is a pointer to a string which must contain a command line. The format of the command line is the same as the format for entering the command line at a terminal. The command can be any iRMX II Human Interface command (as described in the *Operator's Guide To The Extended iRMX II Human Interface*) or any command that you write. However, it can not be a CLI command, and it cannot use the alias feature of the CLI.

If the string specified as a parameter to C$SEND$COMMAND contains a complete command invocation, C$SEND$COMMAND places the command line in the command connection and invokes the command.

However, if the string does not contain the entire command invocation (that is, it contains the "&" as a continuation character), C$SEND$COMMAND places the command line in the command connection without invoking the command. It also returns an exception code, E$CONTINUED, to inform the calling program that the command is continued. Additional C$SEND$COMMAND calls place continuation lines in the command connection, combining them with the command lines already there. When C$SEND$COMMAND sends the last portion of the command invocation (a line without a continuation character), it invokes the entire command.

Once you call C$SEND$COMMAND enough times to place a complete command invocation in the command connection, C$SEND$COMMAND invokes the command. This involves loading the command from secondary storage and starting it running. The C$SEND$COMMAND call that invokes the command does not return control until the invoked command finishes processing. Once the command finishes processing, you can use the command connection for invoking other commands.

The C$SEND$COMMAND system call contains two pointers to words that receive iRMX II condition codes. One of these (called except$ptr in the system call description) points to a word that receives the status of the C$SEND$COMMAND system call. An E$OK indicates that C$SEND$COMMAND received the full command invocation and invoked the command. An E$CONTINUED indicates that the command invocation is not complete (the last line contained a continuation character). Other exception codes indicate other problems with the system call.

The other pointer (called command$except$ptr in the system call description) points to a word that receives the status of the invoked command. This allows you to determine the status of the invoked command.

5.4 PRIORITY CONSIDERATIONS

Every command has a priority (usually based on the priority of the user who invoked the command) that determines when the command will be able to run in relation to the other tasks in the system. When commands are invoked via command connections, their priorities are lowered (numerically increased) by one. This ensures that the calling task (the one that created the command connection) retains control over the commands it invokes.

As a result, a command invoked directly at the terminal will have a higher priority (and possibly complete sooner) than the same command invoked via a command connection.

5.5 DELETING THE COMMAND CONNECTION

After you have finished invoking commands programmatically, you should delete the command connection. The C$DELETE$COMMAND$CONNECTION system call performs this operation. You do not need to delete the command connection after each command invocation, because the command connection is reusable. However, you should delete the command connection after performing all C$SEND$COMMAND operations. This frees the memory used by the data structures of the command connection.

5.6 EXAMPLE

Figure 5-1 contains an example of a program that uses:

C$CREATE$COMMAND$CONNECTION
C$SEND$COMMAND
C$DELETE$COMMAND$CONNECTION

It invokes the Human Interface COPY command programmatically.

```
/*************************************************************************
*                                                                        *
*  This example demonstrates the use of the following Human Interface   *
*  functions:                                                            *
*                                                                        *
*  rq$C$create$command$connection                                        *
*  rq$C$send$command                                                     *
*  rq$C$delete$command$connection                                        *
*                                                                        *
*  This program uses the previous system calls to invoke the command    *
*  COPY :F1:OLD to :F1:NEW and then continues normal processing.        *
*  The program is invoked with the command line:                        *
*                                                                        *
*      PROG2                                                             *
*************************************************************************/

prog2: DO;

DECLARE token LITERALLY 'SELECTOR';

$include (/rmx286/inc/error.lit)
$include (/rmx286/inc/hi.ext)
$include (/rmx286/inc/eios.ext)

DECLARE (ci$token, co$token, command$connection$token)  TOKEN,
       (excep, comexcep, exexcep)  WORD;
DECLARE output$prep  BYTE;

                    .
                    .
                    .

/* Invoke utility to copy file OLD to file NEW */

/* Get tokens for CI and CO */
ci$token = rq$C$get$input$connection(@(4,':CI:'), @excep);
IF excep <> E$OK       THEN
   CALL rq$exit$io$job (excep, NIL, @exexcep);
co$token = rq$C$get$output$connection(@(4,':CO:'), output$prep, @excep);
IF excep <> E$OK       THEN
   CALL rq$exit$io$job (excep, NIL, @exexcep);
```

Figure 5-1. Command Connection Example (continued)

```
/* Create command connection */
command$connection$token = rq$C$create$command$connection (ci$token,
                                                          co$token, 0,
                                                          @excep);
IF excep <> E$OK       THEN
   CALL rq$exit$io$job (excep, NIL, @exexcep);

/* Send command to copy files */
CALL rq$C$send$command (command$connection$token,
                        @(23,'COPY :F1:OLD TO :F1:NEW'),
                        @comexcep, @excep);
IF excep <> E$OK       THEN
   CALL rq$exit$io$job (excep, NIL, @exexcep);

/* Delete command connection */
CALL rq$C$delete$command$connection (command$connection$token, @excep);
IF excep <> E$OK       THEN
   CALL rq$exit$io$job (excep, NIL, @exexcep);
                     .
                     .       Rest of program
                     .
/* Finish I/O processing */
CALL rq$exit$io$job (excep, NIL, @exexcep);

END prog2;
```

Figure 5-1. Command Connection Example

CHAPTER 6
PROGRAM CONTROL

6.1 OVERVIEW

Normally, when a Human Interface command is executing an operator cannot communicate with the command (or with the application system in general) unless the command initiates the communication by requesting input from the terminal. This can present problems if an operator inadvertently enters the wrong command, or if the operator decides while the command is executing that the command is unnecessary. Under these circumstances, there are a number of ways the operator can abort command execution.

- If the command is executing interactively, the operator can enter a Control-C character to abort a command.
- If the command is running in the background environment (explained in Chapter 2), the operator can enter the CLI commands JOBS and KILL to abort a job.

This chapter explains how to override the default Control-C mechanism by providing your own code to process a Control-C character. For more information on aborting background jobs, see the *Operator's Guide To The Extended iRMX II Human Interface.*

6.2 HOW THE DEFAULT CONTROL-C MECHANISM WORKS

When the operator enters a Control-C, the Operating System sends a unit to a semaphore. In the default case, it sends the unit to a semaphore established by the Human Interface. A Human Interface task waits at that semaphore to receive the unit. When it receives the unit, it aborts the command that is currently executing and returns control to the operator. The Human Interface task then waits at the semaphore for another unit.

This Control-C facility allows operators to cancel commands while the commands are executing. It is a valuable facility that can be used with your commands without requiring you to provide special implementation code.

6.3 PROVIDING YOUR OWN CONTROL-C MECHANISM

With some commands that you write, you might want to override the default Control-C mechanism. For example, suppose you write a text editor. An operator invokes the editor with a Human Interface command and then specifies edit commands to enter text into a buffer and modify that text.

While using the editor, the operator does not want a Control-C character to abort the entire editing session, destroying text in the editing buffer that may have taken hours to create. Instead, the operator might want a Control-C to abort a single editor command only. In order to provide this facility, your Human Interface command (the editor) must override the default Control-C mechanism and provide its own code to handle Control-C entries.

To override the default Control-C mechanism, you must change the semaphore to which the Operating System sends the unit when the operator enters a Control-C. By changing the semaphore to one that you create, you circumvent the Control-C task of the Human Interface. You can use the Human Interface system call C$SET$CONTROL$C to replace the Control-C semaphore. This system call changes the calling job's Control-C semaphore to the semaphore you specify. There is only one parameter in this system call: control$C$semaphore which is a token for your new Control-C semaphore. A single unit is sent to the new semaphore each time a Control-C is entered from the terminal. A complete description of C$SET$CONTROL$C is in the *Extended iRMX II Human Interface System Calls Reference Manual.*

If your command replaces the default Control-C semaphore with its own, it should also service that semaphore. It can do this either by creating a task that waits continually at the semaphore for a unit or by containing in-line code that periodically checks the semaphore.

In either case, when a unit is sent to the semaphore, the command (or the task) must perform the necessary Control-C operation.

The program flow of such a command would be:

1. Call CREATE$SEMAPHORE to create the Control-C semaphore.
2. If you plan to create a Control-C task to service the semaphore, call CATALOG$OBJECT to catalog the token for the semaphore in an object directory.
3. If you plan to use a Control-C task, have the program call CREATE$TASK to start the Control-C task.
4. Call C$SET$CONTROL$C to switch the Control-C semaphore to the one just created. Use the token for the semaphore you created in Step 1 as input.

5. Continue with command processing. If you are servicing the Control-C semaphore in-line, periodically check the semaphore (by calling RECEIVE$UNITS with the no wait option) to determine if it contains any units. If you obtain any units from the semaphore, perform the necessary Control-C processing.

To service the Control-C with a task, the program flow of the Control-C task could be:

1. Call LOOKUP$OBJECT to obtain the token for the semaphore.
2. Do forever:
   a. Call RECEIVE$UNITS (with the wait forever option, 0FFFFH) to obtain a unit from the semaphore.
   b. Perform the operation that must occur when the operator enters a Control-C.

Each method of servicing the Control-C semaphore has advantages and disadvantages.

If your code services the Control-C semaphore with in-line code, you can perform any operation you want. You can branch to various locations, you can start new tasks running, you can abort the command, or you can perform any other function that you wish. However, in order to service the Control-C semaphore with in-line code, you must check the semaphore periodically, to see if it contains a unit. When doing this, you must ensure that you place the checks inside all program loops that perform operations an operator might want to abort. Also, because you can check the semaphore only periodically, you cannot always guarantee a quick response to the Control-C.

If you use a Control-C task, you can guarantee quick service because the task is always waiting at the semaphore. However, because a separate task services the Control-C, you can perform only a limited number of operations in response to the Control-C.

- The task can send a message to the command, but then the command would have to periodically check a mailbox. This has the same disadvantages as in-line servicing with none of the advantages.
- The task can delete or suspend the command. However, the task has no way of knowing what operations the command was performing when the operator entered the Control-C. If the command was updating an internal table, deleting the command could corrupt your entire system. Suspending the command could allow the Control-C task to interrogate the command's state. The Control-C task could delete the command if appropriate, or it could allow the command to run until it was safe to be deleted.

Once your command assigns a new Control-C semaphore, the semaphore remains assigned until either of the following things occur:

- Your command invokes the Human Interface C$SEND$COMMAND system call. Invoking this system call automatically reverts back to the default Control-C mechanism. To continue using your own Control-C mechanism, invoke

C$SET$CONTROL$C (to switch back to your Control-C semaphore) immediately after invoking C$SEND$COMMAND.

- Your command is deleted. When this happens, the Human Interface automatically reactivates its default Control-C semaphore. For example, once the example text editor described earlier in this chapter terminates, the Human Interface resets the semaphore so that Control-C again becomes active.

6.4 A SAMPLE CONTROL-C TASK

This section provides an example of a user-supplied Control-C mechanism.

```
/***************************************************************************

    TITLE:     control$C$handler

    ABSTRACT:
        This task waits at a semaphore for a single unit.
        If a unit is received, the current job is terminated.

    CALLING SEQUENCE:

        CALL rq$create$task( ..., @control$C$handler, ...);

    ALGORITHM:

        LOOKUP$OBJECT for the Control-C semaphore (should be cataloged
                                                  under 'SEMA');
        DO FOREVER;
            wait for unit at Control-C semaphore;
            if a unit is received, terminate the job using exit$io$job;
        END;

***************************************************************************/
```

Figure 6-1. A CONTROL-C Task Example (continued)

```
hcctask: DO;

$subtitle('control$C$handler')


control$C$handler:    PROCEDURE
                     REENTRANT  PUBLIC;

DECLARE token LITERALLY 'SELECTOR';

$include (/rmx286/inc/error.lit)
$include (/rmx286/inc/hi.ext)
$include (/rmx286/inc/eios.ext)

                        /* local variables */
    DECLARE
        E$OK                 LITERALLY   '0',
        local$excep            WORD,
        units              WORD,

        INFINITE$WAIT      LITERALLY   'OFFFFH',
        sema                 TOKEN;

    sema = rq$lookup$object( SELECTOR$OF(NIL, @(4,'SEMA'),INFINITE$WAIT,
                                @local$excep);

    IF  local$excep <> E$OK    THEN
        CALL rq$exit$io$job(local$excep,NIL,@local$excep);

    DO FOREVER;
       units-rq$receive$units(sema,1,INFINITE$WAIT,
                               @local$excep);
       IF  local$excep = E$OK  THEN
            CALL rq$exit$io$job(0,NIL,@local$excep);
    END;

END control$C$handler;

END hcctask;
```

Figure 6-1. A CONTROL-C Task Example

CHAPTER 7 CREATING HUMAN INTERFACE COMMANDS

7.1 OVERVIEW

This chapter discusses the steps that you must perform to create your own Human Interface commands. It discusses the necessary elements of a command as well as how to compile (or assemble) and bind your code (using BND286).

To perform the operations described in this chapter, you must have either an 8086-based Microcomputer Development System (such as a Series IV) or an iRMX II-based system that includes the Human Interface commands. Either system must have an editor, the necessary compiler or assembler, and the utility programs (such as BND286).

7.2 ELEMENTS OF A HUMAN INTERFACE COMMAND

This section discusses the rules that every user written command must obey. It also suggests some programming practices to make coding and using your commands easier.

NOTE

When coding your commands, be careful not to duplicate CLI command names such as, ALIAS and SUBMIT. If you give a command a CLI command name, you must execute it with the full pathname, for example, :util:alias. Otherwise, the CLI command will be executed instead of your command.

7.2.1 Parsing The Command Line

If you are going to allow the operator to enter parameters when invoking the command, the first thing your command should do is parse the command line. Chapter 3 describes the Human Interface system calls that you can use. To support lists of pathnames and wild-carded pathnames, the flow of a program that uses input and output files should be:

1. Call C$GET$INPUT$PATHNAME to obtain the entire list of input pathnames.
2. Call C$GET$OUTPUT$PATHNAME to obtain the preposition and the entire list of output pathnames.

3. Call C$GET$PARAMETER as many times as necessary to get all the parameters.
4. Do until no more input pathnames remain:
   a. Call C$GET$INPUT$CONNECTION to obtain a connection to the input file.
   b. Call C$GET$OUTPUT$CONNECTION to obtain a connection to the output file.
   c. Read the information from the input file, perform the command operations based on that input, and write the information to the output file.
   d. Call S$DELETE$CONNECTION (Extended I/O System call) to delete the connections to the input and output files.
   e. Call C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME to obtain the next input and output pathnames.

7.2.2 Avoiding The Use of Certain System Calls

When you write the code for your Human Interface command, you can use any of the iRMX II system calls, depending on the requirements of your command. However, some system calls are intended primarily for use in system-level jobs (those jobs that you configure into the Operating System rather than invoking as Human Interface commands). In the descriptions of system calls, the iRMX II system call reference manuals contain cautions concerning those system calls that you should avoid using.

In particular, avoid iRMX II objects (and their associated system calls) that, by their use, make your command immune to deletion. Regions and extension objects (described in the *Extended iRMX II Nucleus User's Guide*) are examples of such objects. If your command becomes immune to deletion, a Control-C that an operator enters to cancel the command will have no effect; the operator's terminal may also lock when the command finishes processing.

7.2.3 Terminating The Command

When the operator invokes a command, the Operating System loads the command into memory and creates an I/O job as the environment in which the command runs. (The *Extended iRMX II Extended I/O System User's Guide* discusses I/O jobs.) The operator can use the CLI BACKGROUND command to process commands in background mode, and at the same time continue processing another command in the foreground. In order to finish processing a foreground command correctly, any task in the command that exits must do so by calling EXIT$IO$JOB (an Extended I/O System call, described in the *Extended iRMX II Extended I/O System Calls Reference Manual*). This system call causes the Operating System to delete the I/O job containing the command, therefore returning control to the operator. If the command running in the foreground omits the call to EXIT$IO$JOB, the operator might not be able to enter further commands. To terminate a command before it reaches its normal completion, the operator should enter CONTROL-C to abort a command running in the foreground or the CLI KILL command to abort a command running in the background environment.

7.2.4 Include Files

When writing the code for your commands, you must declare each iRMX II system call as an external procedure. Instead of writing these declarations yourself, you can use the $INCLUDE statement. $INCLUDE statements make it possible to include code from an external file into your program. The following information may be in an $INCLUDE file: external declarations of system calls, literal definitions of exception codes, and common literal definitions that you declare.

The iRMX II $INCLUDE files are created during software installation and are located in the :SD:RMX286/INC directory. There is an $INCLUDE file for each subsystem. This file includes all the external declarations for all the system calls of the subsystem. For example, the $INCLUDE file for the Human Interface is :SD:RMX286/INC/HI.EXT. It contains all the external declarations for the Human Interface system calls. To use these files, simply determine the subsystem that your command requires and code $INCLUDE statements for the corresponding external declaration files into your source program.

You might also require literal definitions of exception codes so that you can refer to the exception codes by their mnemonics instead of by their values (for example, E$MEM instead of 2H). After software installation, the :SD:RMX286/INC directory contains the ERROR.LIT file consisting of LITERALLY declarations. The file defines all the iRMX II condition code mnemonics used. Refer to the *Extended iRMX II Hardware and Software Installation Guide* for information about the release diskettes and the files contained in them. Refer to the *PL/M-286 User's Guide* for information about the $INCLUDE statement.

7.3 PRODUCING AN EXECUTABLE COMMAND

After you have written the source code for your command, you must produce object code that can be executed in an iRMX II environment. This involves the following procedure:

1. Compile (or assemble) the command using the appropriate translators. When you do this, ensure that the names you specify in $INCLUDE statements specify the correct devices and directories.
2. Using BND286, bind the code to iRMX II interface libraries (and any other libraries that you require) and produce a relocatable object module that the Operating System can load anywhere in memory. The format of the BND286 command is:

```
BND286                      &
  command-name,             &
  :dir:other.lib,           &
  :SD:RMX286/LIB/RMXIF*.LIB  &; replace * (C = COMPACT, L = LARGE)
      RCONFIGURE (DYNAMICMEM(min,max)) OBJECT(output-pathname) &
      SEGSIZE(STACK(stacksize))
```

where:

command-name	complete pathname of the file containing your compiled (or assembled) command. You can bind in several files or libraries at this point, if necessary.
:dir:	A generic logical name you create for directories containing miscellaneous libraries.
other.lib	Any other files or libraries that you need to bind with your command, for example, PLM286.LIB.
output-pathname	Complete pathname of the file in which BND286 places the command after binding.
stacksize	Size, in bytes, of the stack needed by the command and any system calls that the command makes. The Human Interface uses this value when it creates a job for the command. Be sure the stack is large enough to handle both user and system requirements. The default value is 1200H. Refer to the *Extended iRMX II Programming Techniques Reference Manual* for information about stack requirements of the system calls.
min max	Minimum and maximum amount of dynamic memory, in bytes, required by the command. The default value for both parameters is zero.

The command uses this memory when it creates iRMX II objects. The Application Loader uses the min and max values when it loads a job for the command. Be sure that these values are large enough to satisfy the needs of your command and small enough to allow the command to be loaded into the operator's memory partition. For example, suppose a sort command requires at least 64K bytes of dynamic memory but can use any additional dynamic memory for buffers to increase performance. If you do not define a maximum memory parameter, all of your dynamic memory will be allocated to the sort command, preventing you from executing other commands at the same time. Therefore, assume that you want to limit the max value to 1M byte. You should specify:

```
RCONFIGURE(DYNAMICMEM(10000H,100000H))
```

Be sure to consider the following factors when calculating the values for min and max.

- The value you give for min and the memory required by the Human Interface program must fit into contiguous memory. If there is not enough contiguous memory for them, you may not be able to load your command.

- The max value should be large enough to ensure enough memory for commands that request memory dynamically.

The command is now ready for execution. An operator can invoke the command by entering the pathname of the file containing the command (the output-pathname in the BND286 command).

If you are using an Intellec Microcomputer Development System to compile or bind your command, you must connect the development system to your iRMX II application system via the iSDM monitor and use the Human Interface UPCOPY command to copy the bound command from the development system disk to an iRMX II secondary storage device. The UPCOPY command is described in the *Operator's Guide To The Extended iRMX II Human Interface*. After you transfer the bound command to an iRMX II secondary storage device, an operator can invoke the command by entering its pathname at the iRMX II terminal.

CHAPTER 8 CONFIGURATION OF THE HUMAN INTERFACE

8.1 OVERVIEW

The Human Interface is a configurable layer of the Operating System. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, Intel provides three kinds of information:

- A list of configurable options
- Detailed information about the options
- Procedures to allow you to specify your choices

The rest of this chapter describes the configurable options. For information on the second and third categories, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

Human Interface configuration consists of two parts: resident configuration and nonresident configuration.

Resident configuration involves configuring the portion of the Human Interface that resides in system memory at all times. This configuration takes place during the configuration of the entire Operating System, when you run the Interactive Configuration Utility to adjust parameters, include or exclude layers of the Operating System, and generate an executable version of the Operating System. You cannot change the resident configuration without reconfiguring the entire Operating System.

Nonresident configuration involves setting up an iRMX II directory structure and placing information about terminals and users into iRMX II files. The nonresident configuration information must be present when the application system starts running, but it can be modified while the system is running. Changes to terminal configuration take effect the next time you initialize your application system. Changes to user configurations take place whenever the affected users logon to the application system.

8.2 RESIDENT CONFIGURATION

When you perform the resident Human Interface configuration, you can modify parameters of the Human Interface that affect all Human Interface users. These include:

- Information about the Human Interface's initial job, such as minimum and maximum memory pool size and whether jobs created by the Human Interface expect to use the 80287 Numeric Processor Extension.
- Information about the resident user (if applicable), including terminal name, terminal device, user ID, maximum priority, pathname of initial program, and default directory. The resident user can be a normal resident user who has control as soon as the system is booted or a recovery resident user who gains control only when initialization errors occur in the configuration files. A system can have only one resident user.
- Information about the jobs created by the Human Interface, including minimum and maximum memory pool sizes.
- Initial size of the buffer that the Human Interface uses when constructing commands.
- Maximum length of a command pathname.
- List of directories that the Human Interface automatically searches, in order, when trying to find a command.
- Pathname of the directory assigned to the logical name :SYSTEM: and a list of pathnames and the logical names that you want the Human Interface to assign upon initialization.
- Information about the default resident initial program including whether the Human Interface uses the Intel supplied CLI as its default initial program or a customized CLI. If a customized CLI is included, you must also specify its pathname.
- Information about user extensions. If you specify the Intel-supplied CLI as the resident initial program, you can specify the pathname of the CLI user extension that is to be incorporated. The default CLI does not contain user extensions.

8.3 NONRESIDENT CONFIGURATION

The nonresident configuration involves specifying information about the terminals and users that access a multi-access Human Interface.

For each terminal in the system, you must specify:

- Terminal device
- Terminal type (name)
- The user name associated with a static logon terminal

For each user in the system, you should specify:

- Logon name
- User ID
- Encrypted Password

- Minimum and maximum memory partition sizes
- Default directory, whose pathname is assigned to the logical name :HOME:
- Pathname of the initial program (optional)
- Maximum job priority

8.4 INITIALIZATION ERROR REPORTING

During the configuration process, you can elect to have the system report Human Interface initialization errors. If you respond "Yes" to the Report Initialization Errors (RIE) parameter on the "Nucleus" screen, the Operating System reports initialization errors from all subsystems. On encountering a Human Interface initialization error, the Operating System returns control to the iSDM monitor after writing the following message to the iSDM console:

```
Human Interface Initialization Error: <error code number>
```

If Report Initialization Errors is not configured into your system or the iSDM monitor is not present, the initial Human Interface task places the Human Interface ID code (4) and the corresponding error code into the first two words of the Nucleus data segment (1E0:0000H). It then goes into an infinite loop.

A complete list of iRMX II error codes can be found in Appendix A of the *Operator's Guide To The Extended iRMX II Human Interface*.

APPENDIX A HUMAN INTERFACE TYPE DEFINITIONS

A.1 TYPE DEFINITIONS

The Extended iRMX II Operating System recognizes these data types:

BYTE	An unsigned, eight-bit, binary number.
WORD	An unsigned, 16-bit, binary number.
DWORD	An unsigned, 32-bit binary number, occupying two contiguous words of memory.
INTEGER	A signed, two-byte, binary number stored in two's complement form.
POINTER	Two words containing the segment selector and an offset into that segment. The offset is in the WORD with the lowest address.
SELECTOR	A 16-bit quantity that is equivalent to the selector portion of a POINTER.
TOKEN	A word containing the logical address of an object. Tokens are selectors that reference an entry in a descriptor table. The entry in the descriptor table contains the physical address of the object.
STRING	A sequence of consecutive bytes having the structure: `length BYTE, chars (255) BYTE;` The first byte contains the length of the string (the number of succeeding bytes). The subscript of the chars field (255) is the maximum number of bytes in any string. Note, that some system calls limit strings to lengths shorter than 255 bytes. A zero count specifies a null string.
STRING$TABLE	A count byte followed by a sequence of consecutive strings. The value contained in the count byte is the number of strings in the rest of the string table. Since the string table contains only a single byte in which to store the count, the maximum number of strings that a string table can contain is 255. A zero count specifies a null string table.

APPENDIX B STRING TABLE FORMAT

B.1 STRING FORMAT

The Extended iRMX II Operating System uses structures called strings to store groups of ASCII characters (such as pathnames). The Operating System assumes a string to be a series of consecutive bytes broken into two portions: a count byte and text bytes. The first byte in the string is the count byte; its value is set to the number of bytes in the text portion of the string. The text bytes contain the substance of the string.

The Operating System also uses another structure called a string table. A string table consists of a count byte and a series of consecutive strings. As with the string, the first byte in the string table is the count byte; its value is set to the number of strings in the string table. The diagram in Figure B-1 shows the string$table parameter format.

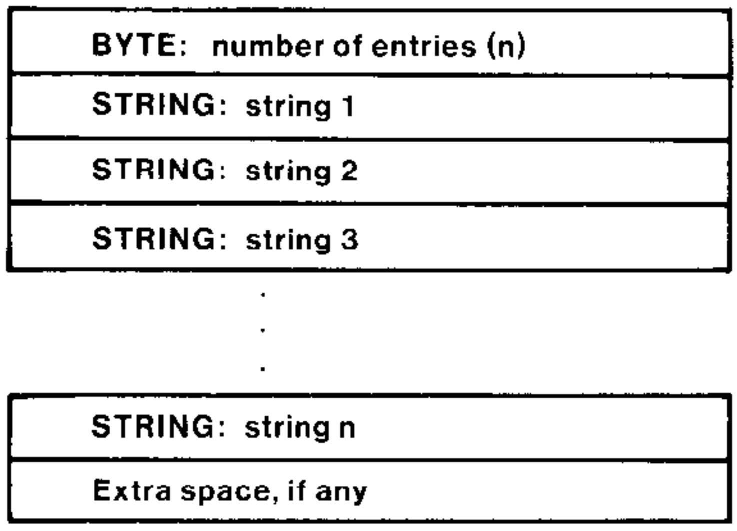


Figure B-1. String Table Format

EXAMPLE:

Assume you wish to generate a string table containing the words HAPPY, GLAD, and SAD. The following declarations would be needed:

```
DECLARE
    p$table(*) BYTE DATA(3,          /* NUMBER OF STRINGS */
        5,'HAPPY',
        4,'GLAD',
        3,'SAD' );
```

INDEX

$INCLUDE statement 7-3
* 1-6
? 1-6

A

aborting command execution 6-1
accessing the values in R?ERROR 2-5
actions of the Human Interface when started 1-1
AFTER preposition 4-2
ALIAS 1-2
ampersand character (&) 3-4
avoiding system-level commands in Human Interface commands 7-2

B

binding a Human Interface Command, example 7-4
binding a user extension 2-10
BND286 7-1, 4

C

C$BACKUP$CHAR 3-6
C$CREATE$COMMAND$CONNECTION 5-1
C$FORMAT$EXCEPTION 4-5
C$GET$CHAR 3-6
C$GET$INPUT$CONNECTION 4-1
C$GET$INPUT$PATHNAME 3-5
C$GET$OUTPUT$CONNECTION 4-2
C$GET$OUTPUT$PATHNAME 3-5, 11
C$GET$PARAMETER 3-5
C$SEND$CO$RESPONSE 4-3
C$SEND$COMMAND 5-2
C$SEND$EO$RESPONSE 4-4
C$SET$CONTROL$C 6-2
CATALOG$OBJECT 6-2
circumflex 3-20
CLI extensions 2-4
command connection 5-1

command connection, creating 5-1
command line interpreter (CLI), description of 1-1
command line, priority 5-3
command name, obtaining 3-20
command$except$ptr 5-3
command-line structure 3-1
command-name 3-2
comment character 3-4
communicating with the operator's terminal 4-3
configuring the Human Interface 8-1
continuation character 3-4
continuation character 5-2
continuation mark for CLI commands 2-3
Control-C 6-1
CREATE$SEMAPHORE 6-2
CREATE$TASK 6-2
creating a command connection 5-1
creating I/O connections1
creating user extensions 2-4
custom CONTROL-C mechanism 6-2
custom initial program 2-11

D

data types A-1
deleting a command connection 5-3
Device and volume management commands, list of 1-2
dynamic logon terminals 1-4
dynamic memory partitions 1-4

E

E$CONTINUED 5-2
elements of a Human Interface command 7-1
error handling in the CLI 2-5
example
  BND286 7-4
  CONTROL-C task 6-4
  string B-2
  user extension 2-5
example code using command connections 5-3
example program flow for a CONTROL-C mechanism 6-2
example program flow for a CONTROL-C task 6-3
examples
  COPY3

FORMAT3
quoting characters 3-5
UPCOPY 3-3
EXIT$IO$JOB 2-12
EXIT$IO$JOB 7-3
extensions, of the CLI 2-4

F

features of the CLI
Aliasing 2-1
Background processing 2-2
I/O redirection 2-2
Line-editing 2-1
Session history 2-2
Set 2-2
features of the Human Interface 1-2
File management commands, list of 1-2
format of a string B-1
formatting messages based on exception codes 4-4

G

General Utility commands, list of 1-2

H

HISTORY 1-2
how CONTROL-C works 6-1
how the CLI parses command lines 2-3
how to create user extenstions 2-4
Human Interface Calls
Command-parsing system calls 1-3
I/O and message processing system calls 1-3
Human Interface CallsCommand-processing system calls 1-3
Human Interface CallsProgram control system call 1-3
Human Interface, configuring 8-1
Human Interface, features of 1-2

I

include files 7-3
initial job 8-2
initial program
standard 1-5

initial program 1-1, 2
initial program 2-11
initialization error reporting 8-3
initialization of the CLI 2-2
inpath-list 3-2
input and output connections 4-1
invoking CLI commands 2-3
iRMX-NET 1-4

J

JOBS 6-1

K

keyword = value-list or keyword (value-list) 3-13
keyword value-list 3-3, 13
keyword(value-list) 3-3
keyword=value-list 3-3
KILL 6-1

L

LAN support 1-4
Line terminator character 3-3
logging off a terminal 1-5
logical name
  SYSTEM 8-2
logon 1-3
logon facility 1-3

M

multi-access support 1-5
multiple terminal support 1-5

N

network access 1-4
Nonresident configuration 8-1, 2
nonresident information for each user in a system
  Encrypted Password 8-2
  default 8-3
  Logon name 8-2
  Maximum job priority 8-3
  Minimum and maximum memory partition sizes 8-3
  Pathname of the initial program (optional) 8-3
  User ID 8-2
nonresident information for terminals
  The user name associated with a static logon terminal 8-2
nonresident user 1-3
nonstandard command lines 3-17

O

obtaining the command name20
outpath-list 3-3
OVER preposition 4-2

P

parameters 3-3
parameters in a BND286 command 7-4
parsing input and output pathnames 3-6
parsing nonstandard command lines 3-15
parsing other parameters 3-12
parsing the command line 7-1
parsing the command line 3-5
pathnames
  wild-card 1-6
preposition
  after 3-2
  over 3-2
  to 3-2
preposition 3-2
priority of the command line 5-3
purposes of C$GET$PARAMETER 3-12

Q

quoting characters 3-5

R

R?ERROR 2-5
RECEIVE$UNITS 6-3
Resident configuration 8-1
resident Human Interface Commands 1-2
resident user 1-3
restrictions on custom initial programs 2-12
rules for custom initial program 2-11
rules for using GET$INPUT- and GET$OUTPUT$PATHNAME 3-7

S

semicolon character (4
SET 1-2
standard initial program 1-5
static logon terminals 1-4
steps for invoking commands from a program 5-1
string example B-2
string format B-1
structure
  message for error codes 4-5
switching to another parsing buffer 3-18

T

terminals
  dynamic logon 1-4
  logging off 1-5
  static logon 1-4
  support for multiple 1-5
terminating a Human Interface command 7-3
three procedures of a CLI extenstion
  epilogue 2-5
  initialization 2-4
  processing 2-4
types of data A-1
typical programming scenario 4-2

U

user
- non-resident 1-3
- resident 1-3

user extension
- binding 2-10

user extension example 2-5
using the command connection 5-2

V

value-list 3-3, 12
variations of the standard command line 3-16

W

wild card characters 3-10
wild cards
- asterisk 3-10
- question mark 3-10

wild-cards 1-6
writing an executable command 7-4

intel®

EXTENDED iRMX® II APPLICATION LOADER USER'S GUIDE

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual explains the operation of the Extended iRMX II Application Loader. Application programmers who want to load programs from secondary storage under the control of extended iRMX II tasks should read this manual. This manual explains only the general operations of the Application Loader. For detailed documentation on extended iRMX II Application Loader system calls, consult the *Extended iRMX II Application Loader System Calls Reference* manual.

READER LEVEL

This manual assumes knowledge of the iRMX II Operating System and terminology associated with it.

CONVENTIONS

All iRMX II system calls begin with one of two standard prefixes: RQ$ or RQE$. When referring to the system calls that begin with RQ$, this manual uses a shorthand notation and omits the prefix. For example, S$OVERLAY means RQ$S$OVERLAY. The actual PL/M-286 external procedure names used to invoke these system calls are shown only in the *Extended iRMX II Application Loader Reference* manual, which lists the detailed calling sequences.

When referring to system calls that begin with RQE$, this manual spells out the complete names, including the RQE$ characters.

There are some system calls whose names are identical except for the RQ$ or RQE$ prefix (for example, the Application Loader system calls RQ$A$LOAD$IO$JOB and RQE$A$LOAD$IO$JOB). The difference between two similarly named system calls is that the RQ$ version operates as it did under the iRMX 86 Operating System and is available for compatibility. The RQE$ version is updated to support new iAPX 286 features, such as 16M byte memory pools. Unless compatibility with iRMX 86 systems is an issue, Intel recommends that you use the system call with the RQE$ preface instead of the one with the RQ$ preface.

CONTENTS

CHAPTER 1
INTRODUCTION TO THE iRMX® II APPLICATION LOADER
1.1 Overview 1-1
1.2 Application Loader Features 1-2
1.2.1 Device Independence 1-2
1.2.2 Synchronous And Asynchronous System Calls 1-2
1.2.3 Support For Overlaid Programs 1-2
1.2.4 Configurability 1-2
1.3 Preparing Code For Loading 1-2
1.3.1 Segsize Control 1-4
1.3.2 Dynamicmem Option 1-4

CHAPTER 2
APPLICATION LOADER SYSTEM CALLS
2.1 Overview 2-1
2.2 RQ$ And RQE$ System Calls 2-1
2.3 I/O Job And Non-I/O Job System Calls 2-1
2.3.1 I/O Job System Calls 2-2
Pool Size for the New Job 2-2
2.3.2 Non-I/O Job System Calls 2-3
2.4 Synchronous And Asynchronous System Calls 2-3
2.4.1 Synchronous System Calls 2-3
2.4.1.1 RQ(E)$S$LOAD$IO$JOB 2-3
2.4.1.2 S$OVERLAY 2-4
2.4.2 Asynchronous System Calls 2-4
2.4.2.1 Example of Asynchronous Calls 2-5
2.4.2.2 Generalities Concerning Asynchronous Calls 2-6
2.4.3 Response Mailbox Parameter 2-6
2.5 EXAMPLE 2-8

APPENDIX A
CONDITION CODES
A.1 Overview A-1
A.2 Normal Condition Codes A-2
A.3 Programmer Error Codes A-2
A.4 Environmental Condition Codes A-2

TABLES

Table 1-1. iRMX® II Stack Size.. 1-4

CHAPTER 1
INTRODUCTION TO THE iRMX® II APPLICATION LOADER

1.1 OVERVIEW

The Extended iRMX II Application Loader, a configurable layer of the iRMX II Operating System, loads programs under the control of iRMX II tasks or tasks that are part of application programs.

The Application Loader provides system calls that load programs from secondary storage into memory. Using the Application Loader provides several advantages:

- Allows programs to run in systems with insufficient memory to accommodate all programs at one time.
- Allows programs that are seldom used to reside on secondary storage rather than in memory.
- Makes it easier to add new programs to the system.

The Application Loader also enables you to implement large programs by using overlays. For example, suppose that your application system includes a large data processor. By dividing the data processor into several parts, you can avoid keeping the entire data processor in memory. One of the parts, called the root, remains in RAM as long as the data processor is running. The root uses the Application Loader to load the other parts, called overlays, whenever they are needed.

This chapter will help you understand the capabilities of the Application Loader by providing background information.

After reading this chapter, you should be able to understand the system calls described in the *Extended iRMX II Application Loader System Calls Reference Manual.*

Readers familiar with the iRMX 86 Application Loader may find it helpful to start by reading Appendix B, which describes the main differences between the iRMX 86 and iRMX II Application Loaders.

1.2 APPLICATION LOADER FEATURES

Several features: device independence, synchronous and asynchronous system calls, overlaid programs support, and configurability, make the iRMX II Application Loader valuable in any application system that loads programs from secondary storage into RAM.

1.2.1 Device Independence

The Application Loader can load object code from any device if the device supports iRMX II named files, and if an iRMX II device driver is available. See the *Extended iRMX II Interactive Configuration Utility Reference Manual* for a complete list of Intel supplied device drivers.

1.2.2 Synchronous And Asynchronous System Calls

The Application Loader provides both synchronous and asynchronous system calls. If you want your tasks to explicitly control the overlapping of processing with loading operations, you can use asynchronous system calls. If you prefer ease of use to explicit control, you can use synchronous system calls.

1.2.3 Support For Overlaid Programs

The Application Loader contains a system call explicitly designed to simplify the process of loading overlay modules. By using the S$OVERLAY system call, your program can easily load overlay modules contained in the same overlaid object file.

1.2.4 Configurability

The Application Loader is configurable, that is, you can select the features of the Application Loader to meet your exact needs. For more details see Chapter 3 and the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

1.3 PREPARING CODE FOR LOADING

To process your code so that the Application Loader can load it, first use an 80286 translator or assembler (e.g., PLM286, ASM286) to produce linkable object modules. If your code uses only UDI system calls, it can be in any model of segmentation. If it uses any other layer of iRMX II, the SMALL model of segmentation is not supported.

After translating your code, you must use BND286 to produce a load file ready to be loaded. The load file must be an OMF-286 Single Task Loadable (STL) object file with LODFIX records. This kind of load file is produced by BND286 using the RCONFIGURE control. STL format is the only object code format supported by the Application Loader. LODFIX records contain the locations of the selectors that are to be updated after loading. LODFIX records are necessary because of the way the iRMX II Operating System functions.

BND286 assumes one Local Descriptor Table (LDT) per task, and therefore, assigns descriptor table entries to the application program in successive LDT slots. The iRMX II Operating System cannot ensure that the same descriptor table entries will be allocated to the application program, in fact it allocates GDT slots instead. LODFIX records are required to allow the Application Loader to replace each selector in the object file with the new GDT selector assigned at random by the iRMX II Operating System at load time.

You can use the Human Interface (HI) DEBUG command to determine which GDT slots were allocated for your program. For more details see the DEBUG command in the *Operator's Guide To The Extended iRMX II Human Interface*.

Example:

The following example illustrates how an STL object file can be produced. It assumes that the directory attached as :LANG: contains the compiler and the binder, and that the directory attached as :LIB: contains iRMX II interface libraries. Assume that the source code for the program is located in a PL/M-286 file named MY PROG.P28. This program adheres to the COMPACT model of segmentation, and therefore will be linked to the COMPACT interface library of iRMX II (RMXIFC.LIB). To produce an object module from the code in MY PROG.P28, use the following sequence:

```
PLM286 MY_PROG.P28 COMPACT
BND286 &
    MY_PROG.OBJ, &
    :LANG:PLM286.LIB, &
    :LIB:RMXIFC.LIB  &
    OBJECT(MY PROG) SEGSIZE(STACK(+500H)) &
    RCONFIGURE(DYNAMICMEM(5000H, 10000H))
```

Upon completion, the object module (MY PROG) is ready for loading. The next two sections discuss the SEGSIZE control and DYNAMICMEM option of the RCONFIGURE control in more detail.

1.3.1 Segsize Control

The SEGSIZE control should contain the stack size required by your program and the iRMX II layers used. You must adjust the stack size of your program to accommodate the stack requirements of the highest iRMX II layer used. Table 1-1 lists the stack requirements of each layer. Note that these requirements are in addition to the stack size needed for your program. The value given as the minimum stack size includes the requirements of all lower layers, e.g., if you use Nucleus, BIOS and EIOS, you should add 550 bytes to the stack.

Table 1-1. iRMX® II Stack Size

Extended iRMX II Layer	Minimum Stack Size
Nucleus	250 bytes
BIOS	350 bytes
EIOS	550 bytes
Application Loader	700 bytes
Human Interface	1500 bytes
UDI	1750 bytes

1.3.2 Dynamicmem Option

DYNAMICMEM is an option of the BND286 RCONFIGURE control. The Application Loader ensures that your program has enough dynamic memory once it is loaded and running. To do this, BND286 allows you to specify the amount of memory your program will allocate dynamically. The value specified by MIN is always available for your program, while the value specified by MAX allows your program to borrow from its parent. Note that these terms apply only for programs that are loaded as I/O jobs, that is, using A$LOAD$IO$JOB, RQE$A$LOAD$IO$JOB, S$LOAD$IO$JOB, or RQE$S$LOAD$IO$JOB.

CHAPTER 2
APPLICATION LOADER SYSTEM CALLS

2.1 OVERVIEW

The Application Loader system calls can be divided into three categories:

- RQ$ and RQE$ system calls
- I/O job and non-I/O job system calls
- Synchronous and asynchronous system calls

2.2 RQ$ AND RQE$ SYSTEM CALLS

The Application Loader has two system calls prefaced by RQE that allow you to specify memory pools up to 16M bytes (using DWORDs for pool parameters). For each RQE system call there is a similar call prefaced by RQ. Intel recommends that you use the RQE version to allow you to take advantage of the iRMX II features. The RQ version allows you to specify pools up to 1M byte only (using WORDs for pool parameters). The following are the system call pairs:

RQ$A$LOAD$IO$JOB--RQE$A$LOAD$IO$JOB
RQ$S$LOAD$IO$JOB--RQE$S$LOAD$IO$JOB

(When referring to the system calls that begin with RQ$, this manual uses a shorthand notation and omits the prefix.)

2.3 I/O JOB AND NON-I/O JOB SYSTEM CALLS

An I/O job is a job that provides the environment for using the Extended I/O System (EIOS). A task can use EIOS system calls only if it is running in an I/O job environment. Other than the A$LOAD system call, all Application Loader system calls must be invoked within an I/O job environment.

If you are unfamiliar with I/O jobs, refer to the *Extended iRMX II Extended I/O System User's Guide* for details.

2.3.1 I/O Job System Calls

The following system calls load programs within I/O jobs:

A$LOAD$IO$JOB
RQE$A$LOAD$IO$JOB
S$LOAD$IO$JOB
RQE$S$LOAD$IO$JOB

When one of these system calls is issued, the Application Loader creates an I/O job in which the initial task is a part of the Application Loader that loads the program. The loaded code is another task in the new job. Once the code is loaded, the Application Loader task terminates itself, unless the new program contains overlays. In this case the Application Loader task waits for requests to load new overlays.

Pool Size for the New Job

When creating an I/O job the Application Loader must determine the size of the I/O job's memory pool. The Application Loader uses the following information to compute the size of the memory pool for the new I/O job:

- The pool$min input parameter, as a number of 16-byte paragraphs.
- The pool$max input parameter, as a number of 16-byte paragraphs.
- A Loader configuration parameter specifying the default dynamic memory requirements. (Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for information about configuring the Loader.)
- Memory requirements specified in the target file by using the RCONFIGURE control of BND286 when creating the object file.

The Loader gives you two options for setting the size of the I/O job's memory pool:

1. You can set both pool$min and pool$max to 0. If you do this, the Loader decides how large a memory pool to allocate to the new I/O job. The Loader uses the requirements of the target file and the default memory pool size--established when the system is configured--to make this decision. Unless you have unusual requirements, you should choose this option.
2. You can use either pool$min or pool$max to override the Loader's decision on pool size. If the value you enter in pool$min is less than what is required to load the file, the Application Loader ignores your input and sets pool$min to the minimum amount of memory required by your file.

If you set pool$max to 0FFFFFH, the created I/O job has unlimited memory borrowing from its parent.

You should be aware that the pool size parameters in Application Loader system calls are specified in 16-byte paragraphs; however, the pool parameters in the RCONFIGURE control of BND286 are entered in BYTES.

2.3.2 Non-I/O Job System Calls

A$LOAD does not create an I/O job. Instead, the Application Loader task that loads the program is running in the context of the caller's job. The Application Loader does not create a task for the loaded code, but merely places it in memory. If you want this code to run, you must explicitly create a task for it using the Loader Result Segment that you receive on completion of loading. For more details on the Loader Result Segment, see the *Extended iRMX II Application Loader System Calls Reference Manual.* Because no I/O job is created, you can use A$LOAD in systems that were configured without the EIOS layer.

2.4 SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

The iRMX II Application Loader provides two types of system calls: synchronous and asynchronous. Synchronous calls return control to the calling task after all operations are completed. Asynchronous calls are executed concurrently with your code.

2.4.1 Synchronous System Calls

These system calls return to the caller only after the service has completely finished, that is, when loading finishes or terminates due to an error. In the first case, an E$OK code is returned to the caller via the exception pointer specified when the call is invoked. In the second case, an error code is returned. The synchronous system calls are

S$LOAD$IO$JOB
RQE$S$LOAD$IO$JOB
S$OVERLAY

2.4.1.1 RQ(E)$S$LOAD$IO$JOB

These two system calls load the file specified in the PATH$NAME parameter and create an I/O job as the environment for the loaded code. Either call can immediately start or delay execution of the loaded code, depending on the TASK$FLAGS input parameter. If delayed execution is specified, START$IO$JOB must be called after the Application Loader has successfully returned and you are ready to start the program. Another parameter, RESP$MBOX, serves as the exit mailbox for the newly created I/O job. The EIOS sends an exit message to this mailbox when the loaded program (contained within the newly created I/O job) terminates, by calling EXIT$IO$JOB. For more details see CREATE$IO$JOB, START$IO$JOB and EXIT$IO$JOB in the *Extended iRMX II EIOS System calls Reference* manual.

2.4.1.2 S$OVERLAY

The term overlay refers to logically independent subsections of a program. These subsections are not all present in memory at the same time during program execution. Because only executing program subsections (and not the entire program) use memory space at a given phase of program execution, using overlays can reduce the memory space required for a program to execute.

To correctly create programs with overlays on an Intel system, **you must use OVL286** (the 80286 overlay generator) to produce the object files. The Application Loader assumes that you have followed these rules when writing your overlaid program.

- The root is always present in memory.
- No overlay, except the root, may be present in memory unless its parent is present in memory.
- The only possible request, from any given overlay, is that a descendent (in the tree) overlay be loaded.
- Any previously loaded sibling is no longer accessible once an overlay has been loaded.
- No assumptions are made about the preservation of data across multiple requests to load the same overlay.

S$OVERLAY is used whenever the loaded program requires that a new overlay be present in memory. This call can be used only by an overlaid program. It can be issued by any overlay (including the root) to load any of its descendents. Although OVERLAY is synchronous (i.e., returns only on completion of service), it can be used in conjunction with the asynchronous Application Loader system calls. That is, A$LOAD$IO$JOB can load the file containing the program and S$OVERLAY can load the required overlay into memory.

2.4.2 Asynchronous System Calls

Each asynchronous system call has two parts: sequential and concurrent. The sequential part behaves in much the same fashion as the fully synchronous system calls. It verifies parameters, checks the file to be loaded, and prepares the concurrent part of the system call. If any problem is detected during the sequential part, an error code is returned to the caller via the exception pointer and the concurrent part is not started. If no error is detected, an E$OK code is returned to the caller and the concurrent part is started.

The concurrent part runs as an iRMX II task. The task is made ready by the sequential part of the call and runs only when the priority-based scheduling of the iRMX II Operating System gives it control of the processor. The concurrent part also returns a condition code as part of a result segment that is sent to the response mailbox specified when you invoke an asynchronous Application Loader call.

The asynchronous calls are split into two parts to improve performance. The functions performed by these calls are somewhat time-consuming because they involve mechanical devices such as disk drives. By performing these functions concurrently with other work, the Application Loader allows your application to run while the Application Loader waits for the mechanical devices to respond to I/O requests.

2.4.2.1 Example of Asynchronous Calls

Let's look at a brief example showing how your application can use asynchronous calls. Suppose your application must load a program stored on disk. The application issues the A$LOAD system call to have the Loader load the program into memory. Let's trace the action one step at a time.

1. Your application issues the A$LOAD system call. (Asynchronous calls require that your application specify a response mailbox for communication with the concurrent part of the system call.)
2. The sequential part of the A$LOAD call begins to run. This part checks the parameters for validity.
3. If the operating system detects a problem, it places a sequential exception code in the word to which your except$ptr parameter points. It then returns control to your application. It does not make the Loader task ready.
4. Your application receives control. Its behavior at this point depends on the condition code returned by the sequential part of the system call. Therefore, the application tests the sequential condition code. If the code is E$OK, the application continues running until it must use the program loaded from the disk. At this point your application can take advantage of the asynchronous and concurrent behavior of the Loader. For example, your application can use this opportunity to perform computations.

   If your application finds that the sequential condition code is other than E$OK, the application can assume that the Loader did not make ready a task to perform the function.

   For the balance of this example, you can assume that the sequential part of the system call returned an E$OK sequential condition code.
5. Before your application can use the loaded program, it must verify that the concurrent part of the A$LOAD system call ran successfully. The application issues a RECEIVE$MESSAGE system call to check the response mailbox that the application specified when it invoked the A$LOAD system call.
6. When the result segment is received, and successful loading is indicated, your application can use RQ$CREATE$TASK (using the entry point, data segment, and stack segment, specified in the result segment) to activate the loaded program.
7. When the loaded program is no longer required, your application can delete all the segments that were created for this program by the Application Loader, using the

segment list in the end of the Loader Result Segment, then the result segment itself can be deleted.

2.4.2.2 Generalities Concerning Asynchronous Calls

The foregoing example used a specific system call (A$LOAD) to show how asynchronous calls allow your application to run concurrently with loading operations. Now let's look at some generalities about all iRMX II asynchronous calls.

- All of the asynchronous system calls consist of two parts--one sequential and one concurrent. The Loader activates the concurrent part only if the sequential part runs successfully (returns E$OK).
- Every asynchronous system call requires that your application designate a response mailbox for communication with the concurrent part of the system call.
- Whenever the sequential part of an asynchronous system call returns a condition code other than E$OK, your application should not attempt to receive a message from the response mailbox. No message exists because the Application Loader cannot run the concurrent part of the system call.
- Whenever the sequential part of an asynchronous system call returns E$OK, your application can count on the Loader running the concurrent part of the system call. Your application can take advantage of the concurrency by doing some processing before receiving the message from the response mailbox.
- Whenever the concurrent part of a system call runs, the Loader signals its completion by sending an object to the response mailbox. The precise nature of the object depends on which system call your application invoked. You can find out what kind of object comes back from a particular system call by looking up the call in the *Extended iRMX II Application Loader System Calls Reference Manual.*
- The Loader returns a segment to your application's response mailbox. Your application must delete the segment when it is no longer needed. The Loader uses memory for such segments, so if your application fails to delete the segment, it might run short of memory.

2.4.3 Response Mailbox Parameter

Whenever you issue an Application Loader System call (except to RQ$OVERLAY), a response mailbox parameter is specified. This mailbox has two different functions, depending on the system call used.

For RQ(E)$S$LOAD$IO$JOB this mailbox serves to receive the exit message from the loaded I/O job. This exit message is sent by the Extended I/O System to this mailbox, when the loaded program terminates by issuing the RQ$EXIT$IO$JOB system call. For more details on I/O jobs and how they terminate refer to the *Extended iRMX II Extended I/O System User's Guide.*

When an asynchronous system call is invoked (A$LOAD,A$LOAD$IO$JOB, RQE$A$LOAD$IO$JOB) this mailbox serves to allow the Loader to notify the caller that the concurrent part of the system call is finished. The Loader sends a result segment to this mailbox on completion of the loading process.

In general, the Loader Result Segment indicates the result of the loading operation. The format of a Loader Result Segment depends upon which system call was invoked, details on Loader Result Segments are included in descriptions of the A$LOAD and A$LOAD$IO$JOB system calls in the *Extended iRMX II Application Loader System Calls Reference Manual.*

If you use an Asynchronous I/O job system call (e.g., RQ(E)$A$LOAD$IO$JOB), the same mailbox receives the Loader Result Segment as well as the exit message from the newly created I/O job. Therefore, you can wait at the same mailbox two times, first for the result segment and then for the exit message, exactly in this order.

Avoid using the same response mailbox for more than one concurrent invocation of asynchronous system calls because the Application Loader may return Loader Result Segments in an order different than the order of invocation. On the other hand, it is safe to use the same mailbox for multiple invocations of asynchronous system calls if the following conditions are met:

- One task invokes the calls
- The task always obtains the result of one call via RECEIVE$MESSAGE before making the next call

2.5 EXAMPLE

```
$title ('exmp: example of using the Application Loader')

/********************************************************************** *
*
*   ABSTRACT : This module is an example using two Application Loader  *
*              system calls--RQE$A$LOAD$IO$JOB and RQE$S$LOAD$IO$JOB.  *
*                                                                      *
***********************************************************************/

exmp : DO;

DECLARE TOKEN LITERALLY SELECTOR;

/*      INCLUDES     */

$include (common.lit)
$include (:inc:loader.ext)
$include (:inc:eios.ext)
$include (:inc:nuclus.ext)

/*       STRUCTURES       */

/* Loader Result Segment returned by rqe$a$load$io$job */

DECLARE ALOAD$RES$STRUC LITERALLY 'STRUCTURE (
   return$code      WORD,
   excep$code       WORD,
   job$token        TOKEN,
   msg$len          BYTE,
   section$count    WORD,
   error$sec$typ    BYTE,
   undef$refs       WORD,
   mem$requested    WORD,
   mem$received     WORD )';

/* Exit message sent by EIOS on behalf of an I/O job that calls rq$exit$io$job
*/

DECLARE EXIT$MSG$STRUC LITERALLY 'STRUCTURE (
   return$code      WORD,
   excep$code       WORD,
   job$token        TOKEN,
   msg$len          BYTE,
   msg (89)         BYTE)';

/* Exception handler structure */
```

```
DECLARE EXCEP$HANDLER$STRUC LITERALLY 'STRUCTURE (
   proc$ptr    POINTER,
   mode        BYTE)';

 /*     LITERALS     */

DECLARE
   NO$TIME$LIMIT        LITERALLY '0FFFFH',
   UNLIMITED            LITERALLY '0FFFFFH',
   NO$DELAY             LITERALLY '0',
   DELAY$REQ            LITERALLY '2',
   E$OK                 LITERALLY '0',
   TERMINATION$OK       LITERALLY '0100H'
   FIFO                 LITERALLY '0';

/*    VARIABLES      */

DECLARE
   (conn, aload$mbox, sload$mbox, aload$job, sload$job)    TOKEN,
   (aload$res$t, exit$seg$t)   TOKEN,
   (pool$min, pool$max)    DWORD,
   prio            BYTE,
   (status, job$flags, task$flags) WORD;
DECLARE
   aload$res$seg   BASED aload$res$t ALOAD$RES$STRUC,
   exit$msg        BASED exit$seg$t  EXIT$MSG$STRUC,
   excep$handler   EXCEP$HANDLER$STRUC;

/************************************************************************
 * Initialize exception handler structure and create mailboxes for two *
 * calls to the Application Loader.                                     *
 ************************************************************************/

excep$handler.proc$ptr = NIL;
excep$handler.mode = 0;

aload$mbox = rq$create$mailbox (FIFO, @status);
IF status <> E$OK THEN GOTO exit;

sload$mbox = rq$create$mailbox (FIFO, @status);
IF status <> E$OK THEN GOTO exit;
```

```
/************************************************************************
 * Now call rqe$a$load$io$job.  First, obtain a connection to the file,  *
 * then prepare the input parameters.  Let the Application Loader decide *
 * how large a pool the job will have. Whatever decision the Application *
 * Loader makes, it will not allow the new job to borrow memory from     *
 * its parent, i.e., will set max equal to its min. The loaded code will *
 * start execution as soon as it is in memory and will have the maximum  *
 * priority of its parent.                                               *
 ************************************************************************/

conn = rq$s$attach$file(@(7,'my prog'),@status) ;
IF status <> E$OK THEN GOTO exit;

pool$min, pool$max = 0;
prio = 0;
job$flags = 0;
task$flags = NO$DELAY;

 aload$job = rqe$a$load$io$job (conn, pool$min, pool$max, @excep$handler,
                      job$flags, prio, task$flags, aload$mbox,  @status);
IF status <> E$OK THEN GOTO exit;


/************************************************************************
 * Since rqe$a$load$io$job is asynchronous, at this point, only          *
 * its sequential part is executed and loading is still in progress.     *
 * Prepare the parameters for rqe$s$load$io$job and call it. Again       *
 * let the Application Loader decide how large a pool to allocate,       *
 * but now the job will have unlimited 'credit' at the parent's pool.    *
 * In this case we want execution of the code to be under our control,  *
 * i.e., delayed. It is most likely that now two Application Loader      *
 * system calls will load the same file concurrently.                    *
 ************************************************************************/

pool$max = UNLIMITED;
task$flags = DELAY$REQ;
aload$job = rqe$s$load$io$job(@(7,'my prog'), pool$min, pool$max,
    @excep$handler, job$flags, prio, task$flags, sload$mbox,  @status); IF status
<> E$OK THEN GOTO exit;


/************************************************************************
 * At this point, it is known that rqe$s$load$io$job finished            *
 * its work, although, nothing is yet known about rqe$a$load$io$job.     *
 * So we wait at the specified mailbox.                                  *
 ************************************************************************/

aload$res$t = rq$receive$message(aload$mbox, NO$TIME$LIMIT, NIL, @status ); IF
status <> E$OK THEN GOTO exit;
```

```
/************************************************************************
 * At this point the Loader Result Segment can be inspected to determine *
 * the memory pool size allocated to the job, or if an error            *
 * occurred, see its nature etc.                                         *
 ************************************************************************/

IF aload$res$seg.return$code <> TERMINATION$OK THEN GOTO exit;


/************************************************************************
 * It is now known that rqe$a$load$io$job completed successfully, and    *
 * the loaded program is already waiting to get hold of the CPU since    *
 * no delay was requested. The second copy of the program is waiting in  *
 * memory for permission to start, so let it start.                      *
 ************************************************************************/

CALL rq$start$io$job (sload$job, @status);


 /***********************************************************************
 * The two loaded programs are on and running, first wait for them to   *
 * terminate (issue an rq$exit$io$job call), then kill them.            *
 ***********************************************************************/

exit$seg$t = rq$receive$message (aload$mbox, NO$TIME$LIMIT, NIL, @status);
IF status <> E$OK THEN GOTO exit;

/* Here we can examine the exit massage... */

CALL rq$delete$job (aload$job, @status);
IF status <> E$OK THEN GOTO exit;

exit$seg$t = rq$receive$message(sload$mbox, NO$TIME$LIMIT, NIL , @status);
IF status <> E$OK THEN GOTO exit;

CALL rq$delete$job (sload$job, @status);
IF status <> E$OK THEN GOTO exit;

exit:

CALL rq$exit$io$job (0, NIL, @status);
IF status <> E$OK THEN                  /* We are in big trouble... */
   CAUSE$INTERRUPT(3);

/***********************************************************************
 * The end, if an error was detected in this module, recovery can be   *
 * attempted, something can be printed to the terminal or the program  *
 * can just terminate.                                                 *
 ***********************************************************************/

END exmp;
```

CHAPTER 3
CONFIGURING THE APPLICATION LOADER

The Extended iRMX II Application Loader is a configurable layer of the operating system. The iRMX II Operating System enables you to configure the type of load function required by your system. Your system may be configured for a load job, which includes all the Application Loader system calls, or for load, which includes only the A$LOAD system call. If you choose all Application Loader system calls, the EIOS will be incorporated into your system by the ICU.

You can configure the Application Loader read buffer size to optimize loading time by increasing or decreasing the amount of memory used by this buffer. That is, a smaller buffer size may cause a longer load time.

Finally, you can configure the minimum memory pool size. This value is used by the Application Loader while creating an I/O job for newly loaded programs. If you specify zero in the pool minimum parameter, the Application Loader computes the required size.

For more information see the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

APPENDIX A CONDITION CODES

A.1 OVERVIEW

The Extended iRMX II Application Loader uses two kinds of condition codes to inform your tasks of any problems that occur during the execution of a system call--sequential condition codes and concurrent condition codes. The difference between the two kinds of codes involves the method the Application Loader uses to return the code to the calling task.

The meaning of a specific condition code depends on the system call that returns the code. For this reason, this appendix does not list interpretations. Refer to the *Extended iRMX II Application Loader System Calls Reference* manual for an interpretation of the codes.

This appendix provides the numeric value associated with each condition code the Application Loader can return. To use the condition code values in a symbolic manner, you can use the code values Intel supplies in the file ERROR.LIT. Alternatively, you can assign (using the PL/M-286 LITERALLY statement) a meaningful name to each code.

The following three sections correlate the name of a condition code with the value returned by the Application Loader. One section lists the normal condition code, one lists exception codes indicating a programming error, and one lists exception codes resulting from environmental conditions. No distinction is drawn between sequential and concurrent errors because most of the codes can be returned as either.

These sections cover only the condition codes returned by the system calls of the Application Loader. Additional condition codes can be produced by other layers that are used by the Application Loader while loading the file. For example, if a mistake is made in the pathname of the load file, you will get an E$FNEXIST error code issued by the I/O system. A complete list of exception codes can be found in the *Extended iRMX II Nucleus User's Guide*.

A.2 NORMAL CONDITION CODES

NAME OF CONDITION	HEXADECIMAL VALUE
E$OK	0H

A.3 PROGRAMMER ERROR CODES

NAME OF CONDITION	HEXADECIMAL VALUE
E$JOB$PARM	8060H

A.4 ENVIRONMENTAL CONDITION CODES

NAME OF CONDITION	HEXADECIMAL VALUE
E$NOT$CONFIGURED	08H
E$BAD$HEADER	62H
E$EOF	65H
E$NO$LOADER$MEM	67H
E$NO$START	6CH
E$JOB$SIZE	6DH
E$OVERLAY	6EH
E$LOADER$SUPPORT	6FH

INDEX

80286 translator or assembler 1-2

A

A$LOAD 2-3
advantages of using the Application Loader 1-1
Application Loader
    three categories of system calls 2-1
Application Loader features
    configurability 1-2
    device independence 1-2
    support of programs with overlays 1-2
    synchronous and asynchronous system calls 1-2
Application Loader features 1-2
asynchronous calls, facts about 2-6
asynchronous system calls 2-4

B

BND286
    dynamicmem option 1-4
    segsize control 1-4

C

COMPACT interface library 1-3
COMPACT model of segmentation 1-3
condition codes
    environmental A-2
    normal A-2
    programmer errors A-2
condition codes A-1

E

ERROR.LIT A-1
example
  BIND 1-3
  how to produce an Single Task Loadable (STL) file 1-3
example of asynchronous calls 2-5
EXIT$IO$JOB system call 2-3

F

features of the Application Loader 1-2

G

GDT slots 1-3

H

Human Interface (HI) DEBUG command 1-3

I

I/O job calls
  A$LOAD$IO$JOB 2-2
  RQE$LOAD$IO$JOB 2-2
  S$LOAD$IO$JOB 2-2
I/O job calls 2-1

L

Loader configuration parameter 2-2
Loader result segment 2-5
Local Descriptor Table (LDT) 1-3
LODFIX records 1-3

M

memory pool size for a new job 2-2

N

non-I/O job calls 2-1, 3
non-I/Ojob calls
  A$LOAD 2-3

O

OVL286 (the 80286 overlay generator) 2-4

P

pool$max 2-2
pool$max input parameter 2-2
pool$min input parameter 2-2
preparing code for loading by the Application Loader 1-2

R

RCONFIGURE 2-2
RECEIVE$MESSAGE system call 2-5
RESP$MBOX parameter 2-3
response mailbox parameter 2-6
RMXIFC.LIB 1-3
RQ(E)$LOAD$IO$JOB 2-3
RQE$LOAD$IO$JOB 2-2

S

S$OVERLAY, description 2-4
sequential and concurrent parts of asynchronous calls 2-4
SMALL model of segmentation 1-2
stack sizes required for each layer 1-4
START$IO$JOB 2-3
synchronous and asynchronous calls 2-3
synchronous system calls
  RQE$S$LOAD$IO$JOB 2-3
  S$LOAD$IO$JOB 2-3
  S$OVERLAY 1-3

T

TASK$FLAGS input parameter 2-3
The difference between RQ$ and RQE$ system calls 2-1

U

use of LITERALLY to give a name to condition codes A-1

intel

EXTENDED iRMX® II UDI USER'S GUIDE

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual describes Intel's Universal Development Interface as it applies to the Extended iRMX® II Operating System. The manual includes a brief introduction to the UDI and its relationship to the iRMX II Operating System.

CONTENTS

CHAPTER 1 — PAGE
INTRODUCTION TO THE UNIVERSAL DEVELOPMENT INTERFACE
1.1 Overview 1-1
1.2 Example 1-2

CHAPTER 2 — PAGE
UDI SYSTEM CALLS IN AN iRMX® II ENVIRONMENT
2.1 Overview 2-1
2.2 Overview Of UDI System Calls 2-1
2.2.1 Memory Management System Calls 2-1
2.2.2 File-Handling System Calls 2-3
2.2.3 Program Control Calls 2-3
2.2.4 Utility and Command-Parsing Calls 2-4
2.2.5 Condition Codes And Exception-Handling Calls 2-4
2.3 Making UDI Calls From PL/M-286 And ASM286 Programs 2-7
2.3.1 Example PL/M-286 Calling Sequence 2-7
2.3.2 Example ASM286 Calling Sequence 2-7
2.4 Writing Portable Programs Using The UDI 2-7

CHAPTER 3. — PAGE
UDI EXAMPLE
3.1 Overview 3-1
3.2 Example Listing 3-1
3.3 Compiling And Binding The Example 3-5
3.4 UDI Interface Libraries 3-5

APPENDIX A. — PAGE
iRMX® II DATA TYPES A-1

APPENDIX B — PAGE
CONDITION CODES B-1

TABLE

Table B-1. Condition Code Ranges B-1

FIGURES

Figure 1-1. The Application-Software-Hardware Model 1-1
Figure 2-1. Chronology of System Calls 2-2

CHAPTER 1
INTRODUCTION TO THE UNIVERSAL DEVELOPMENT INTERFACE

1.1 OVERVIEW

Intel's Universal Development Interface (UDI) is a set of system calls compatible with several of Intel's operating systems. If an application program makes only UDI system calls with no explicit calls to an individual Intel operating system, the application can be transported between operating systems. Figure 1-1 illustrates the relationship between application code, the processing hardware, and the layers of software that lie between.

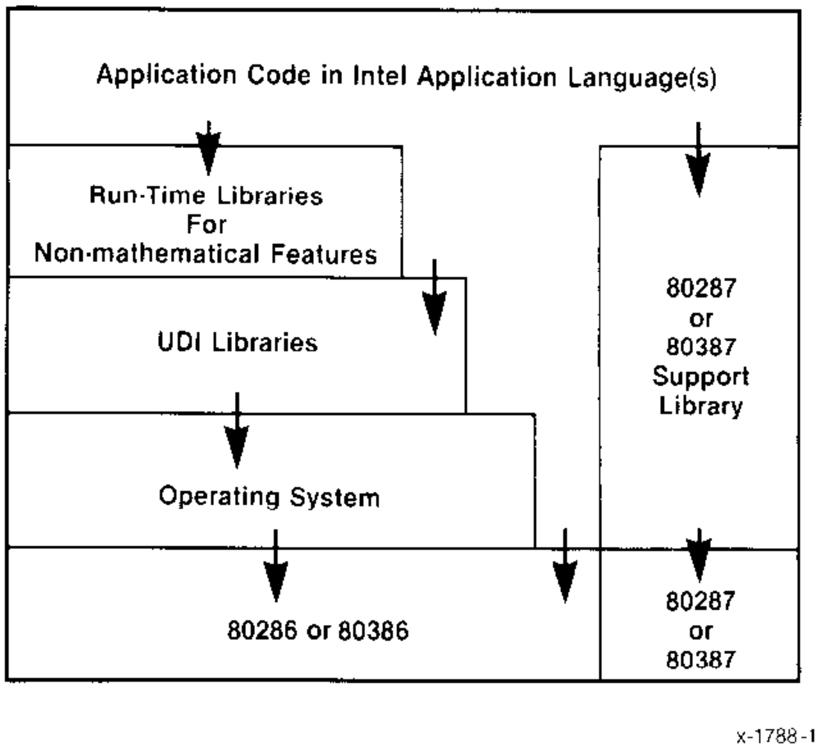

Application Code in Intel Application Language(s)
Run-Time Libraries For Non-mathematical Features
UDI Libraries
Operating System
80286 or 80386
80287 or 80387 Support Library
80287 or 80387
x-1788-1

Figure 1-1. The Application-Software-Hardware Model

1.2 EXAMPLE

In Figure 1-1, the downward arrows represent command flow and data flow from the application code down to the hardware, where the commands are ultimately executed. (Not shown in the figure is another set of arrows showing the upward flow of data from the hardware to the application code.) Note that one of the downward arrows is crossed out, signifying that the application code does not make direct calls to the operating system. Rather, all interaction between the application code and the operating system is done through the UDI software.

the UDI serve as the link between an application and the operating system, you can switch operating systems simply by changing the interface between the UDI and the operating system. In other words, all you need to make an application transportable between operating system environments is a UDI library for each operating system. This library always presents the same interface to the application, but its interface with the operating system is designed specifically and exclusively for that operating system. Intel provides UDI libraries for the iRMX 86, extended iRMX II, Series III, and Series IV operating systems.

The UDI system calls, while presenting a standard interface to user programs, behave somewhat differently when used in different operating system environments. This is because each operating system has many unique characteristics, and some of them are reflected in the results of the UDI calls.

The next chapter discusses the UDI in the context of the Extended iRMX II Operating System.

Intel®

CHAPTER 2 UDI SYSTEM CALLS IN AN iRMX® II ENVIRONMENT

2.1 OVERVIEW

This chapter describes the requirements and behavior of UDI system calls in the Extended iRMX II II environment.

2.2 OVERVIEW OF UDI SYSTEM CALLS

This section discusses the functions of many UDI system calls, highlighting the interrelationships, if any, among calls in various functional groups.

2.2.1 Memory Management System Calls

When the iRMX II Operating System loads and runs a program, the program is allocated memory, in an amount that depends on how the program was configured. The portion of memory not occupied by loaded code and data--the free space pool--is available to the program dynamically, that is, while the program runs. The operating system manages memory as segments that programs can obtain, use, and return.

Programs can use the UDI system calls named DQ$ALLOCATE and $MALLOCATE to get memory segments from the pool. They can use the system calls DQ$FREE and DQ$MFREE to return segments to the pool. They can also call DQ$GET$SIZE and DQ$GET$MSIZE to receive information about allocated memory segments.

NOTE

The system calls DQ$MALLOCATE, DQ$MFREE, and DQ$GET$MSIZE are supported only for programs compiled in the COMPACT or LARGE segmentation models.

The SMALL model of segmentation does not support the long pointers used by these calls to identify the allocated memory.

You can reserve memory for the Extended iRMX II I/O System by using the system call DQ$RESERVE$IO$MEMORY and by declaring the maximum number of buffers and files to be attached at any one time. This action ensures that the operating system allocates sufficient memory to accommodate any I/O buffer needed to open a file. (The I/O System creates buffers for every file it opens.)

DQ$RESERVE$IO$MEMORY is useful for an application program that has used all of its available memory pool and wants to open a temporary file to store the data. If the application program has not previously invoked DQ$RESERVE$IO$MEMORY, the operating system returns an E$MEM exception code when the program tries to create the temporary file.

The E$MEM exception condition occurs because the operating system requires memory from the application job to open and access the specified file. If the application job does not have any available memory, the operating system cannot open or access more files. But if the application program has called DQ$RESERVE$IO$MEMORY previously, the operating system can use the memory reserved by this system call to manipulate the file.

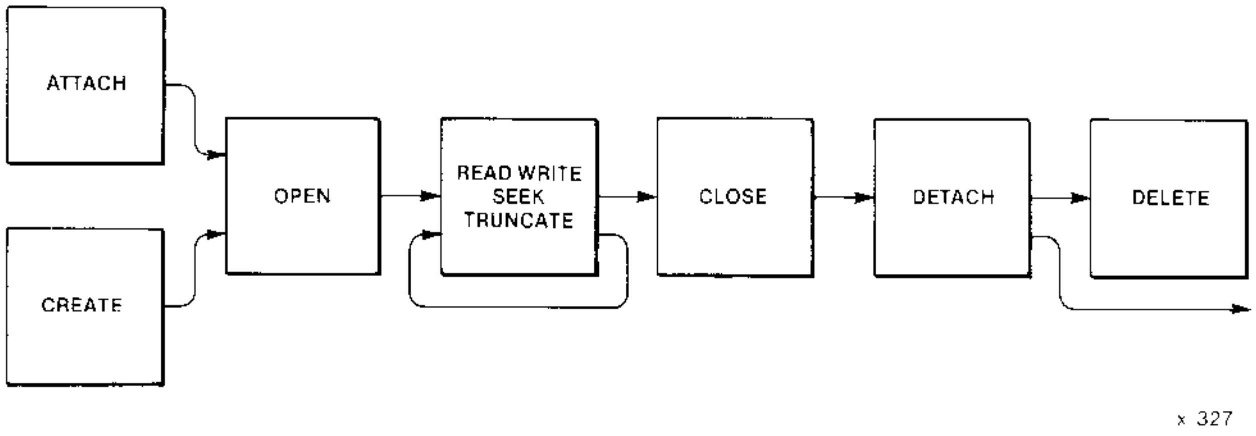


Figure 2-1. Chronology of System Calls

2.2.2 File-Handling System Calls

About one-half of the UDI system calls are used to manipulate files. Figure 2-1 shows the chronological relationships among the most frequently used file-handling system calls.

The key to using Extended iRMX II files is the connection. A program wanting to use a file first obtains (a token for) a connection to the file and then uses the connection to perform operations on the file. Other programs can simultaneously have their own connections to the same file. Each program having a connection to a file uses its connection as if it had exclusive access to the file.

A program obtains a connection by calling DQ$ATTACH (if the file already exists) or DQ$CREATE (to create a new file). When the program no longer needs the connection, it can call DQ$DETACH to delete the connection. To delete both the connection and the file, the program calls DQ$DELETE.

Once a program has a connection, it can call DQ$OPEN to prepare the connection for input/output operations. The program performs input or output operations by calling DQ$READ and DQ$WRITE. It can move the file pointer associated with the connection by calling DQ$SEEK. It can truncate the file by calling DQ$TRUNCATE.

When the program has finished doing input and output to the file, it can close the connection by calling DQ$CLOSE. Note that the program opens and closes the connection, not the file. Unless the program deletes the connection, by calling DQ$DETACH, it can continue to open and close the connection as necessary.

If a program calls DQ$DELETE to delete a file, the file cannot be deleted while other connections and I/O requests attached to the file exist. In that case, the file is marked for deletion and is not actually deleted until the last of the connections is deleted. During the time that it is marked for deletion, no new connections or I/O requests to the file may be issued.

2.2.3 Program Control Calls

UDI provides two system calls for program control: DQ$EXIT and DQ$OVERLAY.

DQ$EXIT terminates a program, closing all open files and freeing all allocated resources. You should always include this system call as the last statement in your program.

DQ$OVERLAY lets you take advantage of the overlay support provided by the operating system. This system call loads an overlay into memory. The overlay must have been prepared by the BND286 binder and the OVL286 overlay generator.

2.2.4 Utility and Command-Parsing Calls

UDI provides system calls for command parsing and for operations such as date and time stamping and system identification. The system calls are DQ$GET$TIME, DQ$DECODE$TIME, DQ$GET$SYSTEM$ID, DQ$GET$ARGUMENT, and DQ$SWITCH$BUFFER.

DQ$GET$TIME and DQ$DECODE$TIME return date and time information as maintained by the operating system. Both system calls provide the same kinds of information, but DQ$DECODE$TIME is a more general system call that should be used instead of DQ$GET$TIME whenever possible.

DQ$GET$SYSTEM$ID returns a string that identifies the name of the operating system. This system call is useful for those programs that need to perform operating-system-specific functions.

DQ$GET$ARGUMENT and DQ$SWITCH$BUFFER enable programs to retrieve parameters from the command line (or from any other program buffer). DQ$GET$ARGUMENT parses the command line, returning the next parameter in the sequence. You can invoke it several times to retrieve all the parameters entered with a command. DQ$SWITCH$BUFFER switches to a new buffer so that the next time you call DQ$GET$ARGUMENT, you will retrieve a parameter from the new buffer.

2.2.5 Condition Codes And Exception-Handling Calls

Every UDI call except DQ$EXIT returns a numeric condition code specifying the result of the call. Each condition code has a unique mnemonic name by which it is known. For example, the code 0, indicating that there were no errors or unusual conditions, has the name E$OK. Any other condition means there was a problem. These conditions are called exceptions.

Exceptional conditions are classified as follows:

- Environmental Conditions. These are generally caused by conditions outside the control of a program; for example, device errors, incorrect file references, or insufficient memory.
- Programmer Errors. These are typically caused by mistakes in programming. Programmer errors can be subdivided into hardware errors and errors in programming. Hardware errors can include such conditions as Divide-by-Zero error, Overflow error, General Protection error, errors detected by the 80287 Numeric Processor Extension (hereafter referred to generically as the NPX), and others. Errors in programming can consist of conditions such as incorrect parameter type, invalid buffer, and others.

If the System Debugger is configured into the application system, all hardware errors will cause the system to break to the monitor. This enables the user to investigate the faulty code using the capabilities of the iSDM 286 monitor and the System Debugger. The break to the monitor will occur regardless of the exception handler and exception mode that are in effect at the time of the exception. If the System Debugger is not present in the application system, UDI handles hardware errors just like other programmer errors.

There is a routine in the UDI interface library called RQ$ERROR that handles UDI exceptions. This routine is called whenever an exception code is generated by a UDI system call. RQ$ERROR performs the following operations:

- If an environmental condition occurs, the exception code is returned to the calling program, which can handle the condition in-line.
- If a programmer error occurs, RQ$ERROR invokes the Nucleus system call SIGNAL$EXCEPTION. The action that SIGNAL$EXCEPTION takes depends on the configuration of the Nucleus (in particular, the setting of the EM parameter of the Nucleus screen in the Interactive Configuration Utility). The EM (exception mode) parameter determines when to transfer control to an exception handler. If the exception mode is NEVER (the default) or ENVIRON, SIGNAL$EXCEPTION passes control back to the calling program so that it can process the exceptional condition in-line. If the exception mode is ALL or PROGRAM, SIGNAL$EXCEPTION passes control to the exception handler that is in effect at the time the exception occurs.

You can override the actions of RQ$ERROR by providing your own RQ$ERROR routine and binding it to your programs. To override the default routine, your routine must contain a PUBLIC procedure named RQ$ERROR, and you must bind the routine to your application code before binding the UDI interface library. That is, in the BND286 command, the name of the file containing your RQ$ERROR routine must appear before the name of the interface library. This causes your RQ$ERROR routine to be bound first, in place of default routine in the interface library. Your RQ$ERROR routine must adhere to the model of segmentation (SMALL, COMPACT, or LARGE) used in the application program itself.

The source code of the default UDI RQ$ERROR routine is available in the /RMX286/UDI directory. You can use this source code as an example when building your own RQ$ERROR routine. The file UCERR.A28 applies to SMALL and COMPACT applications. ULERR.A28 applies to LARGE applications.

As explained earlier, when the RQ$ERROR procedure invokes SIGNAL$EXCEPTION, control can pass to an exception handler. If the default system exception handler (DEF.EXCEPTIONHANDLER) is in effect, it displays the appropriate error message at the console and terminates the program. Your program can establish its own exception handler by calling DQ$TRAP$EXCEPTION. This exception handler will be called whenever RQ$SIGNAL$EXCEPTION is invoked, in the default system this is on programmer error. DQ$DECODE$EXCEPTION returns a mnemonic description of any condition code generated by a UDI system call. The rest of this section provides information that you need to write your own exception handler.

After an exception condition occurs and before your exception handler gains control, the Extended iRMX II Operating System does the following:

1. Pushes the condition code onto the stack of the program that made the system call generating the exception code.
2. Pushes the number of the parameter that caused the exception onto the stack (1 for the first parameter, 2 for the second, etc.).
3. Pushes a WORD onto the stack (reserved for future use).
4. Pushes a WORD for the NPX onto the stack.
5. Initiates a long call to the exception handler.

If the condition was not caused by an erroneous parameter, the responsible parameter number is zero. If the exception code is E$NDP$ERROR, the fourth item pushed onto the stack is the NPX status word, and the NPX exceptions have been cleared.

Programs compiled under the SMALL model of segmentation cannot have an alternate exception handler because alternate exception handlers must have a LONG POINTER, which is not available in the SMALL model. Therefore, programs compiled under the SMALL model must use the default system exception handler.

UDI also provides a method for programs to handle any CONTROL-C characters that are typed while the program is running. The default CONTROL-C handler terminates the program that was active when the CONTROL-C was entered. However, a program can override the default handler for the duration of its execution by calling DQ$TRAP$CC and supplying a long pointer to the new CONTROL-C handler. The operating system will call this new CONTROL-C handler whenever a CONTROL-C is typed at the terminal. The new handler remains in effect until the program calls DQ$EXIT, or until it establishes another handler by calling DQ$TRAP$CC again.

2.3 MAKING UDI CALLS FROM PL/M-286 AND ASM286 PROGRAMS

This section describes how to make UDI calls from a program, using the DQ$ALLOCATE system call as an example. The information from this example will enable you to make the other UDI calls. Two examples are presented: one for a call from a PL/M-286 program and one for a call from an ASM286 program.

This chapter shows the DQ$ALLOCATE system call syntax as follows:

```
seg$token - DQ$ALLOCATE (size, except$ptr);
```

The example that follows requests 128 bytes of memory. It expects a token for the 128-byte segment in ARRAY BASE and a condition code in ERR.

2.3.1 Example PL/M-286 Calling Sequence

```
DECLARE   ARRAY BASE    TOKEN,
          ERR           WORD;
     .
     .
     .
ARRAYBASE = DQ$ALLOCATE (128, @ERR);
```

2.3.2 Example ASM286 Calling Sequence

```
MOV     AX,128
PUSH    AX              ; first parameter
LEA     AX,ERR
PUSH    DS              ; second parameter
PUSH    AX              ;
CALL    DQALLOCATE
MOV     ARRAYBASE,AX  ; returned value
```

This example is applicable to programs that expect to use the COMPACT and LARGE interface to UDI. For the SMALL interface, omit pushing the DS segment register.

2.4 Writing Portable Programs Using The UDI

Not all programs making UDI calls will be portable across all UDI-supported operating systems. However, you can employ the following programming techniques to ensure that the programs you write are portable (or as portable as possible):

- Never examine filenames (and pathnames) in your program. The rules for forming pathnames are operating-system-dependent.

- Modify filename strings only by calling the UDI procedure DQ$CHANGE$EXTENSION.
- Work only with pathnames supplied by the user, pathnames created by calling DQ$CHANGE$EXTENSION, or predefined filenames.
- Always check the exception code to see if a call failed.
- When handling error conditions, you should create the necessary file connections in the initial part of programs or make a DQ$RESERVE$IO$MEMORY call before making any other UDI system call.

CHAPTER 3 UDI EXAMPLE

3.1 OVERVIEW

This chapter presents an example of UDI system calls. Following the program listing is the SUBMIT file that was used to create the executable module.

3.2 EXAMPLE LISTING

```
$compact
$optimize(3)


/*************************************************************************
 *                                                                       *
 * Program UPPER                                                         *
 *                                                                       *
 *  This program demonstrates the use of UDI file-handling and           *
 * command-line-parsing system calls.  The program reads an input        *
 * file of characters and converts all lowercase alphabetic characters   *
 * to uppercase.  The converted data are written to a second file.       *
 *                                                                       *
 * UPPER expects the command line that invokes it to be of the form:     *
 *                                                                       *
 *  UPPER infile [TO outfile]                                            *
 *                                                                       *
 * (If "TO outfile" is not specified, :CO: is assumed.)                  *
 *************************************************************************/


upper: DO;

   DECLARE
       CR              LITERALLY '0DH',
       LF              LITERALLY '0AH',
       TOKEN           LITERALLY 'SELECTOR',
       BOOLEAN         LITERALLY 'BYTE',
       E$OK            LITERALLY '0',
       E$FATAL$EXIT    LITERALLY '3',
       write$only      LITERALLY '2',
       read$only       LITERALLY '1',
       false          LITERALLY '0',
       true           LITERALLY '0FFH';
       co$conn     TOKEN;

$include(/rmx286/inc/udi.ext)
$subtitle('check$exception')

 /*************************************************************************
  *   Procedure to check an exception code.  If the exception code is    *
  *   not E$OK, print a message and exit.                                *
  *************************************************************************/

check$exception:  PROCEDURE (exception, info$p) REENTRANT;
   DECLARE
       exception   WORD,
       info$p      POINTER,
       info        BASED info$p STRUCTURE (
   count   BYTE,
   char(1) BYTE),
       exc$buf STRUCTURE (
                         count        BYTE,
                         char(80)     BYTE),
                         local$excep WORD;

   IF exception <> E$OK THEN
       DO;
           CALL dq$decode$exception (exception, @exc$buf, @local$excep);
           CALL dq$write (co$conn, @exc$buf.char, exc$buf.count, @local$excep);
           CALL dq$write (co$conn, @(': '), 2, @local$excep);
           CALL dq$write (co$conn, @info.char, info.count, @local$excep);
           CALL dq$write (co$conn, @(CR, LF), 2, @local$excep);
           CALL dq$exit (E$FATAL$EXIT);
       END;

END check$exception;
$subtitle('Main')

/*************************************************************************
 *                           --- MAIN PROGRAM ---                        *
 *************************************************************************/


   DECLARE
       single$buffer   LITERALLY '0',
       double$buffer   LITERALLY '2',
       status          WORD;

   DECLARE
       in$name(50)     BYTE,
       out$name(50)    BYTE,
       in$conn         TOKEN,
       out$conn        TOKEN,
        delim           BYTE;

    DECLARE
        buffer(1024)    BYTE,
        in$count        WORD,
        i               WORD,
        not$done        BOOLEAN;


/*********************************************************************
 *          Create a connection to :CO:  (console output)           *
 *********************************************************************/


    co$conn = dq$create (@(4, ':CO:'), @status);
    CALL dq$open (co$conn, write$only, single$buffer, @status);


/*********************************************************************
 *          Ignore the name of the program (the first argument).    *
 *********************************************************************/


    delim = dq$get$argument (@buffer, @status);
    CALL check$exception (status, NIL);
    IF delim = CR THEN
        CALL dq$exit (E$OK);
/*********************************************************************
 *                    Attach the input file, and open it.           *
 *********************************************************************/


    delim = dq$get$argument (@in$name, @status);
    CALL check$exception (status, NIL);

    in$conn = dq$attach (@in$name, @status);
    CALL check$exception (status, @in$name);

    CALL dq$open (in$conn, read$only, double$buffer, @status);
    CALL check$exception (status, @in$name);


/*********************************************************************
 *     Find out if there is an output file specified.  If so, attach *
 *     and open it.  If not, use :CO: for output.                    *
 *********************************************************************/


    IF delim <> CR THEN
        DO;
            delim = dq$get$argument (@buffer, @status);
            CALL check$exception (status, NIL);
            IF  (delim = CR) OR
                (buffer(0) <> 2) OR
                (buffer(1) <> 'T') OR
                (buffer(2) <> 'O')
            THEN
                DO;
                    CALL dq$write (co$conn, @('Invalid output file', CR,
                        LF), 21, @status);
                    CALL dq$exit (E$FATAL$EXIT);
                END;

            delim = dq$get$argument (@out$name, @status);
            CALL check$exception (status, NIL);

            out$conn = dq$create (@out$name, @status);
            CALL check$exception (status, @out$name);

            CALL dq$open (out$conn, write$only, 2, @status);
            CALL check$exception (status, @out$name);
        END;

        ELSE
            out$conn = co$conn;  /* Write to :CO: if no file specified */

/********************************************************************
 *        Read from input, convert, and write to output             *
 ********************************************************************/


            not$done = true;

            DO WHILE not$done;
                in$count = dq$read (in$conn, @buffer, size(buffer),
                    @status);
                CALL check$exception (status, @in$name);

                IF in$count = 0 THEN     /* If no characters are in the, */
                    not$done = false;    /* file, then fail next test.   */

                IF not$done THEN     /* If characters are in the */
                    DO;              /* file, then process them. */

                        DO i=0 TO in$count-1;
                            IF (buffer(i) >= 'a') AND (buffer(i) <= 'z')
                            THEN buffer(i) = buffer(i) + 'A'-'a';
                    END;

                CALL dq$write (out$conn, @buffer, in$count, @status);
                CALL check$exception (status, @out$name);
            END;
        END;
```

```
/*****************************************************************
 *               Close input and output files, and exit          *
 *****************************************************************/


   CALL dq$close (in$conn, @status);
   CALL check$exception (status, @in$name);

   CALL dq$close (out$conn, @status);
   CALL check$exception (status, @out$name);

   CALL dq$exit (E$OK);

END upper;
```

3.3 COMPILING AND BINDING THE EXAMPLE

The program UPPER was compiled and bound on an extended iRMX II-based system using the following commands:

```
plm286 upper.p28  <CR>
bnd286 upper.obj, &  <CR>
:lang:plm286.lib, &  <CR>
/rmx286/lib/udiifc.lib rc(dm(1000H,2000H)) &  <CR>
ss(stack(2000H)) object(upper)  <CR>
```

These commands can be entered directly at the terminal, or placed in a file with a .CSD extension and invoked by using the SUBMIT command.

3.4 UDI INTERFACE LIBRARIES

The interface libraries for the UDI are

UDIIFS.LIB	SMALL model
UDIIFC.LIB	COMPACT model
UDIIFL.LIB	LARGE model

APPENDIX A iRMX® II DATA TYPES

The following data types are recognized by the iRMX 286 Operating System.

BYTE	An unsigned, eight-bit binary number.
WORD	An unsigned, two-BYTE, binary number.
INTEGER	A signed, two-BYTE, binary number. Negative numbers are stored in two's-complement form.
POINTER	Two consecutive WORDs containing the base selector of a (64K-byte processor) segment and an offset into the segment. The offset is in the word having the lower address.
SELECTOR	An index into a descriptor table that identifies a particular memory segment. The descriptor table entry lists the segment's base, its limit, its type, and its privilege level.
TOKEN	A SELECTOR that identifies an object. A token must be declared literally a SELECTOR.
DWORD	A 4-BYTE unsigned binary number.
BOOLEAN	A BYTE that is considered to have a value of TRUE if it is 0FFH, and FALSE if it is 00H. A boolean must be declared literally a BYTE.

APPENDIX B CONDITION CODES

This appendix lists the condition code ranges generated by each layer of the Extended iRMX II Operating System. Exception codes are classified as either "Environmental Conditions" or "Programmer Errors." The latter classification includes certain hardware errors as well as programming errors.

The values of these condition codes fall into ranges based on the Extended iRMX II layer that first detects the condition. Table B-1 lists the layers and their ranges (in hexadecimal). The table shows the layer(s) that could generate the code, in case you wish to refer to the appropriate manual.

Table B-1. Condition Code Ranges

Layer	Environmental	Programming
Nucleus	1H to 1FH	8000H to 801FH
I/O Systems	20H to 5FH	8020H to 805FH
Application Loader	60H to 7FH	8060H to 807FH
Human Interface	80H to AFH	8080H to 80AFH
Universal Development Interface	C0H to DFH	80C0H to 80DFH
Reserved for Intel	E0H to 3FFFH	80E0H to BFFFH
Reserved for users	4000H to 7FFFH	C000H to FFFFH

The *Operator's Guide to the iRMX II Human Interface* gives the value of each code and its associated mnemonic, as well as a short description of its significance.

INDEX

80287 Numeric Processor Extension 2-4

A

Application-software-hardware model 1-1

B

BND286 2-3

C

Closing connections 2-3
Condition codes 2-4
Condition codes B-1

D

Data types A-1
Deletion of a file 2-3
Divide-by-Zero error 2-4
DQ$ALLOCATE 2-1, 7
DQ$ATTACH system call 2-3
DQ$CLOSE system call 2-3
DQ$CREATE system call 2-3
DQ$DECODE$EXCEPTION system call 2-6
DQ$DECODE$TIME system call 2-4
DQ$DELETE system call 2-3
DQ$DETACH system call 2-3
DQ$EXIT system call 2-3, 4, 6
DQ$FREE system call 2-1
DQ$GET$ARGUMENT system call 2-4
DQ$GET$SIZE system call 2-1
DQ$GET$SYSTEM$ID system call 2-4
DQ$GET$TIME system call 2-4
DQ$MALLOCATE system call 2-1
DQ$OPEN system call 2-3
DQ$OVERLAY system call 2-3
DQ$READ system call 2-3
DQ$RESERVE$IO$MEMORY system call 2-2, 8
DQ$SEEK system call 2-3

DQ$SWITCH$BUFFER system call 2-4
DQ$TRAP$EXCEPTION system call 2-6
DQ$TRUNCATE system call 2-3
DQ$WRITE system call 2-3

E

Environmental Conditions 2-4
Example
  ASM286 calling sequence 2-7
  BIND 3-5
  PL/M-286 calling sequence 2-7
Exception
  codes B-1
  handling 2-4

F

File deletion 2-3
File-handling system calls 2-3
Free space pool 2-1

G

General Protection error 2-4
Guidelines for writing portable programs 2-7

I

Interactive Configuration Utility (ICU) 2-5
Interface libraries for the UDI 3-5
ISDM monitor 2-5

M

Making UDI calls from PL/M-286 and ASM286 2-7
Memory management 2-1
Memory management system calls 2-1

N

NPX 2-4

O

Opening connections 2-3

Operating system actions after an error occurs 2-6
Operating systems, switching 1-2
Overflow error 2-4

P

Program control system calls 2-3
Program portability 2-7
Programmer errors 2-4

R

Reserving memory 2-2
RQ$ERROR 2-5

S

Seeking 2-3
Segments 2-1
SIGNAL$EXCEPTION system call 2-5
Switching operating systems 1-2
System calls
  exception handling 2-4
  file-handling 2-3
  memory management 2-1
  program control 2-3
  utility and command-parsing 2-4
System debugger 2-5

T

Temporary file 2-2
Transporting code 1-2
Types of data A-1

U

Utility and command-parsing 2-4

intel®

EXTENDED iRMX® II DEVICE DRIVERS USER'S GUIDE

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

The I/O System is the part of the Extended iRMX II Operating System that enables access to files on peripheral devices. (The term "I/O System" encompasses both the Basic I/O System and the Extended I/O System.) It is implemented as a set of file drivers and a set of device drivers. A file driver provides user access to a particular type of file, independent of the device on which the file resides. A device driver provides a standard interface between a particular device and one or more file drivers. Thus, by adding device drivers, your application system can support additional types of devices. It can do this without changing the user interface, because the drivers remain unchanged.

This manual describes the different types of device drivers supported by the I/O System (common, random access, terminal, and custom). It illustrates the basic concepts of these drivers, and it describes how to write your own drivers of these types. In addition, it discusses each of the Intel-supplied device drivers and provides any special information needed to use those drivers.

Reader Level

To use the Intel-supplied device drivers, this manual assumes you are familiar with these items:

- The PL/M-286 programming language and/or the ASM286 Macro Assembly Language.
- The Extended iRMX II Operating System and the concepts of operating-system tasks, segments, and other objects.
- The I/O System, as described in the *Extended iRMX II Basic I/O System User's Guide* and the *Extended iRMX II Extended I/O System User's Guide.* These manuals document the user interface to the I/O System.
- The configuration process, as described in the *Extended iRMX II Interactive Configuration Utility Reference* manual.

This manual assumes that if you are writing your own device drivers you are a systems-level programmer experienced in dealing with I/O devices. In particular, it assumes that you are also familiar with the following:

- The hardware codes necessary to perform actual read and write operations on your I/O device. This manual does not document these device-dependent instructions.
- Regions, as described in the *Extended iRMX II Nucleus User's Guide.*

Conventions

Whenever this manual describes I/O operations, it assumes that tasks use Basic I/O System calls (such as RQ$A$READ, RQ$A$WRITE, and RQ$A$SPECIAL). Even though not mentioned, the tasks can also use the equivalent Extended I/O System calls (such as RQ$S$READ, RQ$S$WRITE, and RQ$S$SPECIAL) or UDI calls (DQ$READ or DQ$WRITE) to perform the same operations.

Intel®

CONTENTS

CHAPTER 1 INTRODUCTION — **PAGE**

1.1 I/O Devices and Device Drivers 1-3
1.2 Types of Device Drivers 1-4
  1.2.1 Custom Drivers 1-4
  1.2.2 Random Access and Common Drivers 1-6
  1.2.3 Terminal Drivers 1-7
1.3 I/O Requests 1-7

CHAPTER 2 FEATURES OF THE DRIVER INTERFACES — **PAGE**

2.1 Custom Drivers 2-1
  2.1.1 Advantages of Custom Drivers 2-1
  2.1.2 Disadvantages of Custom Drivers 2-2
2.2 Random Access and Common Drivers 2-2
  2.2.1 Features Common and Random Access Devices Can Access 2-3
    2.2.1.1 Interrupt Tasks and Interrupt Handlers 2-3
    2.2.1.2 Request Queue 2-3
    2.2.1.3 Volume Change Notification 2-3
    2.2.1.4 Attach before Access 2-4
    2.2.1.5 Long-Term Operations 2-4
  2.2.2 Special Features for Random Access Devices 2-4
    2.2.2.1 Dividing I/O Requests by Sector or by Track 2-4
    2.2.2.2 Seek Optimization 2-5
    2.2.2.3 Seek Overlap 2-5
    2.2.2.4 Retries 2-5
2.3 Terminal Drivers 2-5
  2.3.1 Terminal I/O 2-6
    2.3.1.1 Raw Input Buffer 2-7
    2.3.1.2 Terminal Support Code Input Buffer 2-7
    2.3.1.3 Difference between Transparent and Flush Mode 2-8
  2.3.2 Controlling Terminal I/O 2-9
    2.3.2.1 Line-Editing Functions 2-9
    2.3.2.2 Controlling Output to a Terminal 2-11
  2.3.3 Software Control Strings 2-12
    2.3.3.1 OSC Sequence 2-12
    2.3.3.2 APC Sequence 2-13
  2.3.4 Modes of Terminal Operation 2-14
    2.3.4.1 Connection Modes 2-16

CHAPTER 2 (continued) **PAGE**

2.3.4.2 Terminal Modes 2-18
2.3.4.3 Using an Auto-Answer Modem with a Terminal 2-44
2.3.4.4 Obtaining Information about a Terminal 2-48
2.3.4.5 Restricting the Use of a Terminal to One Connection 2-49
2.3.4.6 Programmatically Inserting Data into a Terminal's Input Stream 2-50

CHAPTER 3 **PAGE**
INTEL-SUPPLIED DEVICE DRIVERS

3.1 Random Access and Common Drivers 3-2
3.1.1 iSBC® 208 Disk Driver 3-2
3.1.2 Mass Storage Controller (MSC) Driver 3-2
3.1.2.1 iSBC® 215G Features 3-3
3.1.2.2 iSBX™ 217C Features 3-3
3.1.2.3 iSBX™ 218A Features 3-4
3.1.2.4 iSBC® 214 Features 3-5
3.1.3 iSBX™ 218A Disk Driver 3-5
3.1.4 iSBC® 220 SMD Disk Driver 3-6
3.1.5 iSBC® 186/224 Multi-Peripheral Driver 3-6
3.1.6 iSBX™ 251 Bubble Memory Driver 3-7
3.1.7 iSBC® 264 Bubble Memory Driver 3-8
3.1.8 iSBX™ 350 Line Printer Driver 3-8
3.1.9 iSBC® 286/10(A) Line Printer Driver 3-8
3.1.10 SCSI Driver 3-9
3.2 Terminal Drivers 3-9
3.2.1 iSBC® 186/410 Terminal Driver 3-9
3.2.2 Terminal Communications Controller Driver 3-10
3.2.3 iSBC® 534 Terminal Driver 3-11
3.2.4 iSBC® 544A Terminal Driver 3-12
3.2.5 iSBX™ 351 Terminal Driver 3-12
3.2.6 8274 Terminal Driver 3-13
3.2.7 82530 Terminal Driver 3-14
3.3 Custom Drivers 3-14
3.3.1 Byte Bucket Driver 3-14
3.3.2 RAM Driver 3-15

CHAPTER 4 **PAGE**
DEVICE DRIVER PARAMETERS

4.1 I/O System Interfaces 4-1
4.1.1 Device-Unit Information Block (DUIB) 4-1
4.1.1.1 DUIB Structure 4-1
4.1.1.2 Using the DUIBs 4-6
4.1.1.3 Creating DUIBs 4-8
4.1.2 I/O Request/Result Segment (IORS) 4-9
4.2 DUIB and IORS Fields Used by Device Drivers 4-17
4.3 Device Interfaces 4-21

CHAPTER 5 **PAGE**
WRITING COMMON OR RANDOM ACCESS DEVICE DRIVERS
5.1 I/O System Supplied Routines 5-1
5.1.1 Interrupt Task 5-2
5.1.2 Message Task 5-3
5.2 I/O System Algorithm for Calling Driver Procedures 5-5
5.3 Required Data Structures 5-7
5.3.1 Device Information Table 5-8
5.3.2 Unit Information Table 5-13
5.4 Device Data Storage Area 5-15
5.5 Introduction to Procedures Drivers Must Supply 5-16
5.6 Device Initialization Procedure 5-17
5.7 Device Finish Procedure 5-19
5.8 Device Start Procedure 5-19
5.9 Device Stop Procedure 5-21
5.10 Device Interrupt Procedure 5-21
5.11 Procedures Random Access Drivers Must Call 5-26
5.11.1 NOTIFY Procedure 5-26
5.11.2 SEEK$COMPLETE Procedure 5-27
5.11.3 Procedures for Other Long-Term Operations 5-28
5.11.3.1 BEGIN$LONG$TERM$OP Procedure 5-28
5.11.3.2 END$LONG$TERM$OP Procedure 5-29
5.11.3.3 GET$IORS Procedure 5-29
5.12 Formatting Considerations 5-30

CHAPTER 6 **PAGE**
WRITING TERMINAL DRIVERS
6.1 Terminal Support Code 6-1
6.2 Data Structures Supporting Terminal I/O 6-2
6.2.1 DUIB 6-3
6.2.2 Device Information Table 6-4
6.2.3 Unit Information Table 6-9
6.2.4 Terminal Support Code (TSC) Data Area 6-14
6.2.5 Additional Information for Buffered Devices 6-20
6.3 TSC Procedures Terminal Drivers Must Call 6-30
6.3.1 Terminal Mutual Exclusion Procedure 6-30
6.3.2 Terminal Set Output Buffer Size Procedure 6-31
6.4 Procedures Terminal Drivers Must Supply 6-31
6.4.1 Terminal Initialization Procedure 6-32
6.4.2 Terminal Finish Procedure 6-34
6.4.3 Terminal Setup Procedure 6-34
6.4.4 Terminal Answer Procedure 6-36
6.4.5 Terminal Hangup Procedure 6-37

CHAPTER 6 (continued) **PAGE**

6.4.6 Terminal Check Procedure 6-37
6.4.7 Terminal Output Procedure 6-44
6.4.8 Terminal Utility Procedure 6-45
6.5 Procedures' use of Data Structures 6-47

CHAPTER 7 **PAGE**
WRITING A CUSTOM DRIVER
7.1 Initialize I/O Procedure 7-1
7.2 Finish I/O Procedure 7-2
7.3 Queue I/O Procedure 7-3
7.4 Cancel I/O Procedure 7-4
7.5 Implementing a Request Queue 7-5

CHAPTER 8 **PAGE**
HANDLING I/O REQUESTS
8.1 I/O System Responses to I/O System Calls 8-1
8.1.1 Attach Device System Calls 8-1
8.1.2 Detach Device System Calls 8-1
8.1.3 Read, Write, Open, Close, Seek, and Special System Calls 8-2
8.1.4 Cancel Requests 8-2
8.2 Actions Required of the Device Driver 8-2
8.2.1 F$READ--Function Code 0 8-3
8.2.2 F$WRITE--Function Code 1 8-3
8.2.3 F$SEEK--Function Code 2 8-4
8.2.4 F$ATTACH--Function Code 4 8-4
8.2.5 F$DETACH--Function Code 5 8-4
8.2.6 F$OPEN--Function Code 6 8-4
8.2.7 F$CLOSE--Function Code 7 8-5
8.2.8 F$SPECIAL--Function Code 3 8-5
8.2.8.1 FS$FORMAT$TRACK--Subfunction 0 8-5
8.2.8.2 FS$QUERY--Subfunction 0 8-6
8.2.8.3 FS$SATISFY--Subfunction 1 8-6
8.2.8.4 FS$NOTIFY--Subfunction 2 8-6
8.2.8.5 FS$GET$DRIVE$DATA--Subfunction 3 8-7
8.2.8.6 FS$GET$TERMINAL$ATTRIBUTES--Subfunction 4 8-8
8.2.8.7 FS$SET$TERMINAL$ATTRIBUTES--Subfunction 5 8-8
8.2.8.8 FS$SET$SIGNAL--Subfunction 6 8-9
8.2.8.9 FS$RESET--Subfunction 7 8-9
8.2.8.10 FS$READ$FILE$MARK--Subfunction 8 8-10
8.2.8.11 FS$WRITE$FILE$MARK--Subfunction 9 8-10
8.2.8.12 FS$RETENSION$TAPE--Subfunction 10 8-10
8.2.8.13 FS$SET$BAD$INFO--Subfunction 12 8-11
8.2.8.14 FS$GET$BAD$INFO--Subfunction 13 8-11

CHAPTER 9 **PAGE**
CONFIGURING A USER-WRITTEN DEVICE DRIVER
9.1 Adding Drivers With the UDS and ICUMRG Utilities ... 9-1
9.1.1 UDS Utility ... 9-4
9.1.1.1 UDS Input File ... 9-4
9.1.1.2 Device Information Screens ... 9-9
9.1.1.3 Unit Information Screens ... 9-13
9.1.1.4 Device-Unit Information Screens ... 9-15
9.1.1.5 Invoking the UDS Utility ... 9-17
9.1.1.6 UDS Error Messages ... 9-18
9.1.2 ICUMRG Utility ... 9-21
9.1.2.1 UDS Modules Screen in the ICU ... 9-22
9.2 Adding Your Driver as a Custom Driver ... 9-22
9.2.1 Example of Adding an Existing Driver as a Custom Driver ... 9-25
9.2.1.1 Contents of the DUIB.INC file specified in the (DPN) Parameter ... 9-26
9.2.1.2 Contents of the File Specified in the (TUP) Parameter ... 9-30
9.2.1.3 Portion of System Generation SUBMIT File as Changed by this Process ... 9-31

APPENDIX A **PAGE**
RANDOM ACCESS SUPPORT FOR INTERRUPT-DRIVEN DEVICES
A.1 INIT$IO Procedure ... A-1
A.2 FINISH$IO Procedure ... A-5
A.3 QUEUE$IO Procedure ... A-6
A.4 CANCEL$IO Procedure ... A-8
A.5 Interrupt Task (INTERRUPT$TASK) ... A-11

APPENDIX B
RANDOM ACCESS SUPPORT FOR MESSAGE-PASSING DEVICES
B.1 INIT$IO Procedure ... B-1
B.2 FINISH$IO Procedure ... B-4
B.3 QUEUE$IO Procedure ... B-5
B.4 CANCEL$IO Procedure ... B-7
B.5 Message Task (MESSAGE$TASK) ... B-8

APPENDIX C **PAGE**
SUPPORTING THE STANDARD DISKETTE FORMAT

APPENDIX D **PAGE**
INTERPRETING BAD TRACK INFORMATION

TABLES

TABLE PAGE

2-1 Connection Modes 2-17
2-2 Terminal Modes 2-20
2-3 Escape Sequence 2-35
2-4 Terminal Character Sequences 2-40
2-5 Example OSC Sequences for Common Terminals 2-42
2-6 Menu of Control Character Functions 2-44
3-1 Intel-Supplied Device Drivers 3-1
4-1 DUIB and IORS Fields Used by Common Device Drivers 4-18
4-2 DUIB and IORS Fields Used by Random Access Device Drivers 4-19
4-3 DUIB and IORS Fields Used by Custom Device Drivers 4-20
6-1 Uses of Fields in Terminal Driver Data Structures 6-48

FIGURES

FIGURE PAGE

1-1 Connecting File and Device Drivers 1-2
1-2 Device Numbering 1-3
1-3 Device Driver Interfaces 1-5
2-1 Buffers Used in Terminal I/O 2-6
2-2 Composite OSC Sequence Diagram 2-15
2-3 Escape Sequence Translation 2-29
2-4 Escape Sequence Simulation 2-31
4-1 Attaching Devices 4-8
5-1 Interrupt Task Interaction 5-3
5-2 Message Task Interaction 5-4
5-3 How the I/O System Calls the Device Driver Procedures 5-6
5-4 DUIBs, Device and Unit Information Tables 5-8
5-5 Relationships between Random-Access Driver Procedures 5-17
6-1 Software Layers Supporting Terminal I/O 6-3
6-2 TSC Data Area 6-16
7-1 Request Queue 7-6
9-1 Adding Drivers with UDS and ICUMRG 9-3
9-2 Syntax of UDS Input File 9-5
9-3 UDS Device Information Screen for One-Interrupt Terminal 9-10
9-4 UDS Device Information Screen for Two-Interrupt Terminal 9-11

FIGURE (continued) **PAGE**

9-5 UDS Device Information Screen for Interruptless Terminal Devices 9-11
9-6 UDS Device Information Screen for Random Access Device 9-12
9-7 UDS Device Information Screen for MULTIBUS® II Message-Passing Random Access Device 9-12
9-8 UDS Device Information Screen for General Device 9-13
9-9 UDS Unit Information Screen for Terminal Device 9-14
9-10 UDS Unit Information Screen for Random Access Device 9-14
9-11 UDS Unit Information Screen for General Device 9-15
9-12 UDS Device-Unit Information Screen for Terminal Device 9-15
9-13 UDS Device-Unit Information Screen for Random Access Device 9-16
9-14 UDS Device-Unit Information Screen for General Device 9-16
9-15 Example User Devices Screen 9-25
9-16 Computing Device and Device-Unit Numbers and BND286 Information 9-28
9-16 Computing Device and Device-Unit Numbers and BND286 Information 9-29
9-17 Public Declarations Needed for the DINFO and UINFO Tables 9-31
9-18 Portion of the Modified SUBMIT File 9-32
A-1 Random Access Device Driver Initialize I/O Procedure A-3
A-2 Random Access Device Driver Finish I/O Procedure A-6
A-3 Random Access Device Driver Queue I/O Procedure A-8
A-4 Random Access Device Driver Cancel I/O Procedure A-10
A-5 Random Access Device Driver Interrupt Task A-12
B-1 Random Access Device Driver Initialize I/O Procedure B-2
B-2 Random Access Device Driver Finish I/O Procedure B-5
B-3 Random Access Device Driver Queue I/O Procedure B-7
B-4 Random Access Device Driver Message Task B-10
D-1 Format of Bad Track Information D-3

CHAPTER 1 INTRODUCTION

The I/O System is implemented as a set of file drivers and a set of device drivers. File drivers provide the support for particular types of files (for example, the named file driver provides the support for named files). Device drivers provide the support for particular devices (for example, a Mass Storage Controller device driver provides the facilities that enable you to use an iSBC 214 Multi-Peripheral controller to control a Winchester-type drive, a flexible diskette drive, or a tape drive with the I/O System). Each type of file has its own file driver, and each device has its own device driver.

One of the reasons that the I/O System is broken up in this manner is to provide device-independent I/O. Application tasks communicate with file drivers, not with device drivers. This allows tasks to manipulate all files in the same manner, regardless of the devices on which the files reside. File drivers, in turn, communicate with device drivers, which provide the instructions necessary to manipulate physical devices.

When an application task wants to communicate with an I/O device, it must connect the file driver it wants to use with the appropriate device driver. This connection occurs at execution time when the task invokes either the Basic I/O System call A$PHYSICAL$ATTACH$DEVICE or the Extended I/O System call LOGICAL$ATTACH$DEVICE. In both of these calls, the task must specify which file driver and which device driver are being used together. Once this device connection is established, application tasks can manipulate the device according to the file driver assigned to it (named, physical, or stream), without concerning themselves about device specifics. The device connection established earlier enables the file driver to communicate with the device driver and handle the device specifics automatically.

Figure 1-1 shows the levels of communication and how tasks connect file drivers and device drivers to establish device-independent I/O.

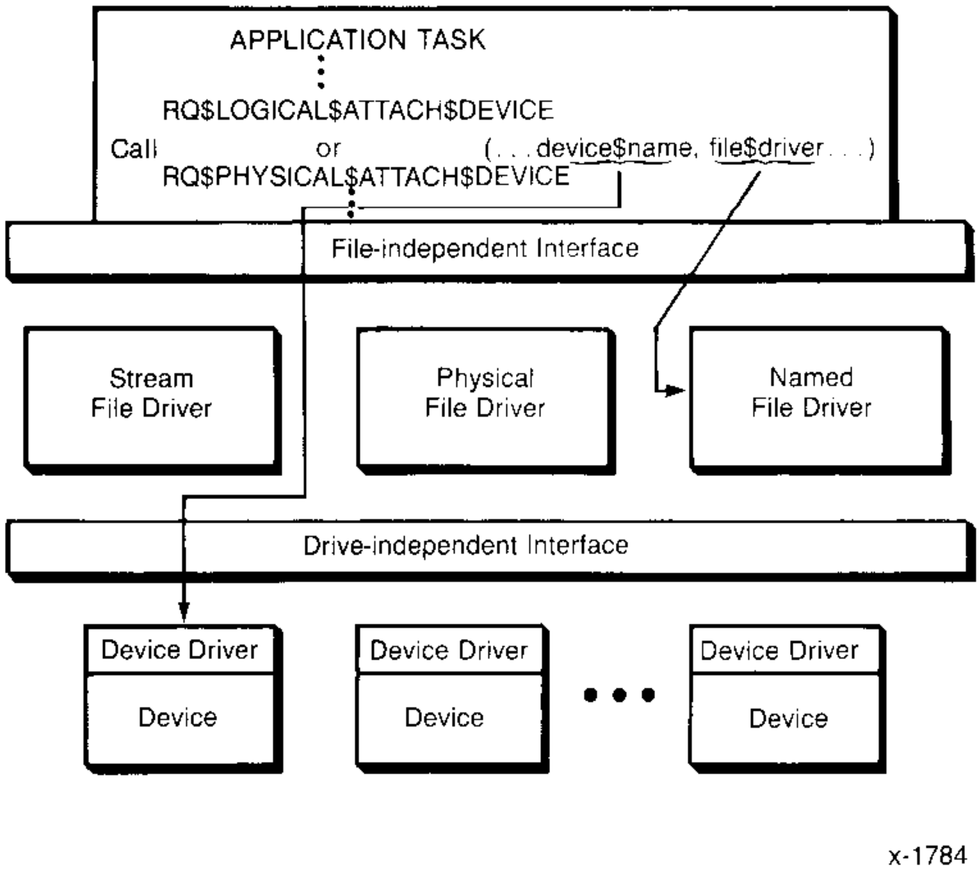


Figure 1-1. Connecting File and Device Drivers

The I/O System provides a standard interface between file drivers and device drivers. To a file driver, a device is merely a standard block of data in a table. To manipulate a device, the file driver calls the device driver procedures listed in the table. To a device driver, all file drivers seem the same. Every file driver calls device drivers in the same manner. This means that the device driver does not need to concern itself with the concept of a file driver. It sees itself as being called by the I/O System, and it returns information to the I/O System. This standard interface has the following advantages:

- The hardware configuration can change without extensive modifications to the software. Instead of modifying entire file drivers when you want to change devices, you need only substitute a different device driver and modify the table.
- The I/O System can support a greater range of devices. It can support any device, as long as you supply a device driver that interfaces to the file drivers in the standard manner.

1.1 I/O DEVICES AND DEVICE DRIVERS

Each I/O device consists of a controller and one or more units. A device as a whole is identified by a unique device number. Units are identified by unit number and by device-unit number. The device number identifies the controller among all the controllers in the system, the unit number identifies the unit within the device, and the unique device-unit number identifies the unit among all the units of all of the devices. Figure 1-2 contains a simplified drawing of three I/O devices and their device, unit, and device-unit numbers.

There must be a device driver (either user-written or Intel-supplied) for every device in the hardware configuration. That device driver must handle the I/O requests for all of the units the device supports. Different devices can use different device drivers; or if they are the same kind of device, they can share the same device driver code. (For example, two iSBC 214 controllers are two separate devices and each has its own device driver. However, these device drivers can share common code.)

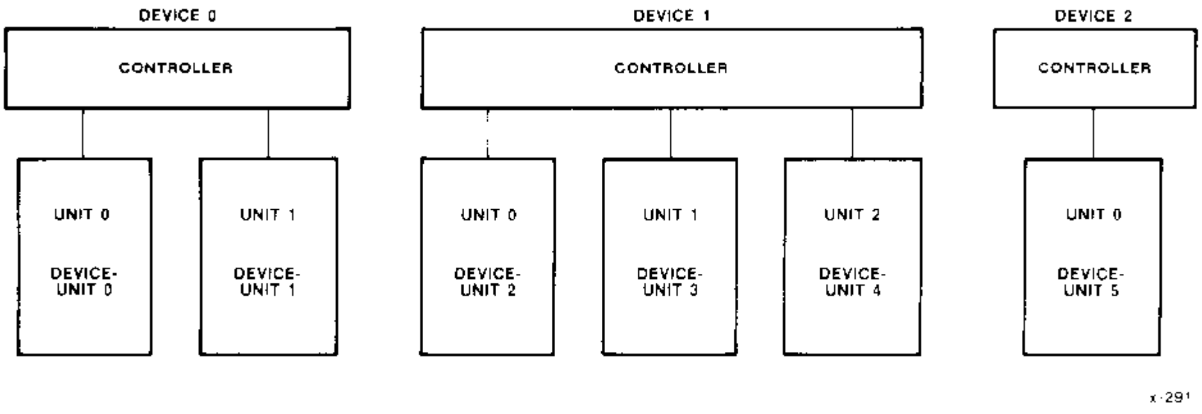


Figure 1-2. Device Numbering

1.2 TYPES OF DEVICE DRIVERS

The I/O System supports four types of device drivers: custom, common, random access, and terminal. These four types are distinguished by whether they have a direct interface to the I/O System or whether they have an interface to Intel-supplied support code. They are also distinguished by which set of support code they use as the interface. Figure 1-3 provides an overview of the interfaces, listing the procedures that the device drivers must supply. The shaded portions of the figure represent the code that the user must write for each type of device driver.

As Figure 1-3 shows, the highest-level interface between the I/O System and the device driver consists of four procedures (known as Initialize I/O, Finish I/O, Queue I/O, and Cancel I/O) that the I/O System calls. Custom drivers must supply these four procedures.

For random-access/common drivers and terminal drivers, the Operating System provides support code that includes these high-level procedures. The support code for each type of driver provides features (such as queuing or baud rate control) needed by each driver of that type. To take advantage of these features when writing random-access/common drivers and terminal drivers, you need to write only a set of lower-level procedures that serve as the interface between the hardware and the support code. Figure 1-3 shows these procedures.

The following sections provide more information about each type of driver.

1.2.1 Custom Drivers

A custom device driver is one you create in its entirety. This type of device driver can assume any form and provide any functions you wish, as long as the I/O System can access it by calling four procedures, designated as

Initialize I/O
Finish I/O
Queue I/O
Cancel I/O

Chapter 2 describes the advantages and disadvantages of writing a custom driver. Chapter 7 describes the interface to which the user-written procedures must adhere.

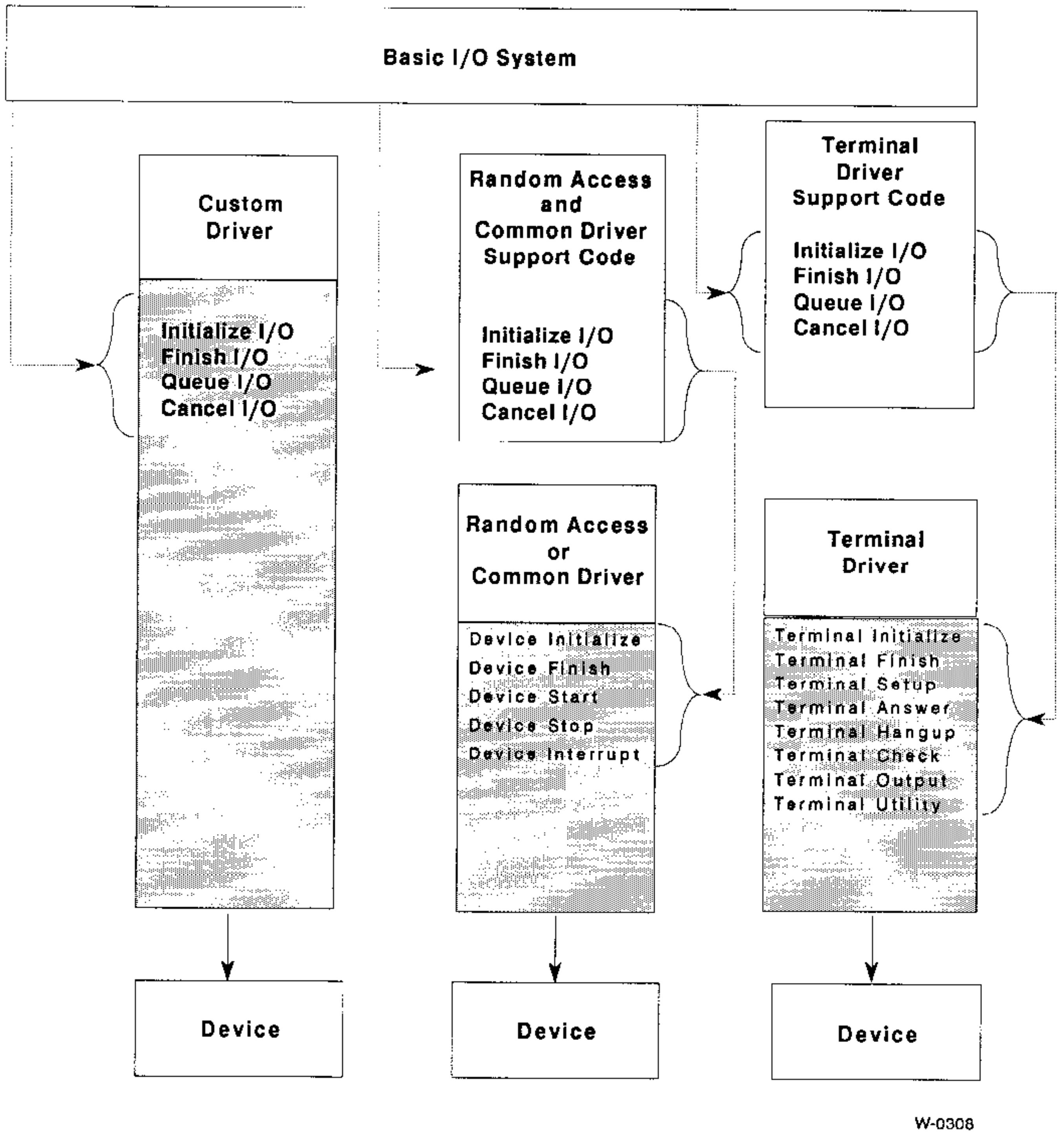


Figure 1-3. Device Driver Interfaces

1.2.2 Random Access and Common Drivers

The Operating System supplies support code that provides much of the high-level code (but not the device-specific code) necessary for operating devices classified as random access or common.

A common device is a relatively simple device such as a line printer (but not a terminal). Devices classified as common devices conform to the following conditions:

- A first-in/first-out mechanism for queuing requests is sufficient for accessing these devices.
- Only one interrupt level is needed to service the device.
- Data either read or written by these devices does not need to be divided into blocks.

A random access device is one in which data can be read from or written to any address of the device (a disk drive, for example). Devices classified as random access devices conform to the following conditions:

- Only one interrupt level is needed to service the device.
- I/O requests must be divided into blocks of a specific length.
- The device supports random access seek operations.

The Operating System provides a single set of support code with the basic routines needed by both common and random access devices. These support routines provide a queuing mechanism, an interrupt handler, and other features needed by common and random access devices. Chapter 2 gives more information about these features.

If you are writing a device driver for a device that fits into either the common or random access classifications, you don't need to write the entire driver. Instead, you can write just the following specialized, device-dependent procedures and set up an interface between them and the support code provided by the Operating System:

Device Initialize
Device Finish
Device Start
Device Stop
Device Interrupt

Chapter 5 describes more about these procedures.

1.2.3 Terminal Drivers

As with common and random access devices, the I/O System provides the high-level support code needed to operate terminals. A terminal is classified as a device that reads and writes single characters or blocks of characters, with an interrupt for each character or block of characters sent.

The features provided by the I/O System's terminal support code are described in Chapter 2. If you are writing a driver for a device classified as a terminal, you don't need to write the entire driver. Instead, you can take advantage of the I/O System's terminal support code by writing just the following device-specific routines:

Terminal Initialize
Terminal Finish
Terminal Setup
Terminal Answer
Terminal Hangup
Terminal Check
Terminal Output
Terminal Utility

Chapter 6 describes more about these procedures.

1.3 I/O REQUESTS

To the device driver, an I/O request is a request by the I/O System for the device to perform a certain function. Functions supported by the I/O System are

Name	Number	Description
Attachdevice	4	Prepare a device for use.
Detachdevice	5	Disconnect a device.
Open	6	Prepare the device or a file on the device for I/O operations.
Close	7	Terminate I/O operations on the device or on a file.
Read	0	Read from the device at the current location.
Write	1	Write to the device at the current location.
Seek	2	Move to a new location on the device (random access devices only).
Special	3	Perform functions that apply to some, but not all, devices. The special functions available include:

Name	Number	Description
Format track	0	Format a track on a mass-storage device.
Query	0	Obtain information about stream-file requests.
Satisfy	1	Artificially satisfy a stream file I/O request.
Notify	2	Request notification when a volume becomes unavailable.
Get disk/ tape data	3	Obtain information about Winchester disks (such as the number of fixed and removable cylinders, the number of sectors, and the sector size) and about tapes (such as whether the tape is present and the number of tracks).
Get terminal data	4	Obtain information about the current configuration of a terminal (such as parity, baud rate, echo mode, and duplex).
Set terminal data	5	Change the current configuration of a terminal.
Set signal	6	Designate a character that, when typed at the keyboard, sends a signal to the system.
Rewind tape	7	Rewind a tape to its load point.
Read tape file mark	8	Move the tape to the next file mark on the tape.
Write tape file mark	9	Write a file mark at the current position on the tape.

Name	Number	Description
Retension tape	10	Fast-forward the tape to the end and then rewind the tape to its load point.
Set bad track/ sector info	12	Write the locations of bad tracks or sectors to a special area of the volume that stores that information.
Get bad track/ sector info	13	Retrieve the locations of bad tracks or sectors from the information stored on the volume.
		Function numbers in the range 14 through 32,767 are reserved by Intel for future use. Function numbers in the range 32,768 through 65,535 decimal might also denote special functions. These numbers are available for user-specified special functions.

The I/O System makes an I/O request by sending to the device driver an I/O request/result segment (IORS) containing the necessary information. (The IORS is described in detail in Chapter 4.) Indicated in the IORS is the type of operation that must be performed, the device, and other important information. The device driver must translate this request into specific device commands to cause the device to perform the requested operation (if the operation is valid for the requested device).

CHAPTER 2 FEATURES OF THE DRIVER INTERFACES

This chapter describes the features available with the different types of drivers. For random access, common, and terminal drivers, this chapter lists the capabilities provided by the I/O System-supplied support code. For custom drivers, for which there is no support code provided, this chapter lists the advantages and disadvantages of writing a complete custom driver instead of adhering to one of the other categories and using the I/O System-supplied support code.

2.1 CUSTOM DRIVERS

The most basic device driver interface is the custom interface. With this interface, the I/O System provides no assistance for standard operations (for example, it doesn't automatically set up a queue to handle device requests). Instead, the custom driver must provide all the functions needed to control the device.

Chapter 7 describes the program interface between a custom driver and the I/O System. The next few paragraphs discuss advantages and disadvantages of writing custom drivers.

2.1.1 Advantages of Custom Drivers

The most obvious advantage of custom drivers is they enable you to add support for devices that don't fit into the common, random access, or terminal categories, and for which Intel doesn't already provide a prewritten driver. The custom driver interface enables you to add support for any device you choose.

Another advantage of custom drivers is they are not restricted to the limitations imposed by the other driver interfaces. For example, the random access support code sets up a queuing mechanism to handle requests for the device. In this queue, requests are handled in a way that minimizes a disk's seek time. If you want to handle device requests based on a priority basis instead, you can write a custom driver to provide that feature.

In summary, the custom driver interface enables you to provide support for any device and in any manner you choose.

2.1.2 Disadvantages of Custom Drivers

There are several disadvantages to writing custom drivers rather than using one of the other driver interfaces.

First, custom drivers usually take longer to write, because you must provide your own versions of the standard features that the I/O System already provides for common, random access, and terminal devices.

Also, debugging time tends to increase for the same reason. With more code to write, there is a greater chance of errors occurring.

Finally, unless you coordinate the design of your custom drivers to allow code sharing, the code size of custom drivers tends to be larger. With random access drivers, all drivers use the same random access support code. With most custom drivers, each driver must provide all of its own functions, thereby duplicating the functions provided by other custom drivers.

2.2 RANDOM ACCESS AND COMMON DRIVERS

The I/O System provides support code that handles many of the standard functions required by common and random access devices. If you use one of the Intel-supplied common or random access drivers, or if you write your own driver and adhere to the common/random access model, your drivers have access to all the capabilities of the I/O System's support code.

The same support code handles both common and random access devices, and the interface to the support code is the same for both kinds of devices (as described in Chapter 6). The I/O System determines whether a device is a common device (like a line printer) or a random access device (like a disk drive) by a value you supply in a table (called a device-unit information block or DUIB) that describes the device to the I/O System. The DUIB is described in detail in Chapter 4.

The value that distinguishes common from random access drivers is called NUM$BUFFERS. A nonzero value for NUM$BUFFERS indicates that the device is a random access device and that the I/O System should use that many buffers of memory to perform data blocking and deblocking operations. These blocking and deblocking operations are special features of the random access support code that guarantee that read and write operations start on sector boundaries.

A zero value for NUM$BUFFERS tells the I/O System that the device is a common device and that there is no need to worry about blocking and deblocking, or about any other special feature associated with random access devices.

2.2.1 Features Common and Random Access Devices Can Access

Several features of the support code are available to both common and random access devices.

2.2.1.1 Interrupt Tasks and Interrupt Handlers

For common and random access devices, the support code creates an interrupt task and an interrupt handler to communicate with the devices. Although the user-written procedures must provide the device-specific code, the support code sets up the structure for the interrupt task and the interrupt handler.

The interrupt level for the handler and task is configurable. You specify the level when you set up the device information table. Refer to Chapter 5 for more information.

2.2.1.2 Request Queue

The support code sets up and manages a queue for handling device requests. Whenever a task requests access to the device, the support code places the request in the queue. It processes requests for common devices in a first-in/first-out manner. It processes requests for random access devices in a modified first-in/first-out manner that minimizes the device's seek time (see the "Seek Optimization" section later in this chapter).

2.2.1.3 Volume Change Notification

One of the options of the A$SPECIAL and S$SPECIAL system calls enables a task to be notified whenever the volume on a device becomes unavailable (such as when swapping diskettes or changing tape cartridges). This feature requires that the device is capable of detecting and signaling volume changes or that the low-level device driver procedures can recognize the change.

After the task requests this notification, the support code signals the task whenever a volume has been changed (or when the stop button has been pressed). This allows the task to detach the device and attach it again, so as not to corrupt the data on a new volume by assuming the previous volume is still present.

If the device itself can recognize when a volume has been changed (or when the stop button has been pressed), all the user-written driver code must do is detect when a door-open or other similar condition occurs and call the I/O System-supplied procedure NOTIFY (described in Chapter 5). The support code handles the rest.

Once the support code notifies an application task that a volume has been changed, the application task can detach the device and reattach it.

2.2.1.4 Attach before Access

If a task attempts to perform an I/O operation on a common or random access device before invoking either the PHYSICAL$ATTACH$DEVICE or LOGICAL$ATTACH$DEVICE system call, the support code recognizes that the device hasn't been initialized. Instead of attempting to perform the operation, the support code returns an E$IO exception code to the invoking task.

2.2.1.5 Long-Term Operations

Some I/O operations (such as rewinding a tape) take a considerable amount of time. Instead of delaying I/O operations on all units of a device while a long-term operation takes place, the support code provides a mechanism that enables drivers to specify when long-term operations start and when they are complete. With this information, the support code can process operations on other units of the same device while the long-term operation is in progress.

For this mechanism to work, the user-written driver code must call the Intel-supplied BEGIN$LONG$TERM$OP and END$LONG$TERM$OP procedures when starting and finishing a long-term operation. Chapter 5 describes these procedures in detail.

Long-term operations on some units require multiple operations. For example, rewinding a tape might require separate rewind and read file mark operations. The support code provides a mechanism that enables drivers to perform multiple operations before notifying the support code that the long-term operation is complete. To use this mechanism, the user-written driver code must call the Intel-supplied GET$IORS procedure each time it wants to perform another intermediate operation. Chapter 5 describes this GET$IORS procedure.

2.2.2 Special Features for Random Access Devices

In addition to the features listed in the previous section, the I/O System's support code supplies several other features that apply specifically to random access devices.

2.2.2.1 Dividing I/O Requests by Sector or by Track

As mentioned earlier, the random access support code can divide I/O requests into multiple requests, each of which reads or writes information starting at sector boundaries. This happens automatically, without special coding in the user-written procedures.

A similar feature enables the I/O System to guarantee that no single I/O request crosses a track boundary on devices with uniform granularity. During device configuration, the user can choose whether the support code provides this feature. When this feature is activated, the support code specifies device addresses as track number/sector number. Otherwise it specifies them just with a logical block number.

2.2.2.2 Seek Optimization

The random access support code automatically orders read, write, and seek requests in a way that minimizes overall device access time. For example, for disk drives, the requests are ordered to minimize head movement. This improves performance of the entire system.

2.2.2.3 Seek Overlap

Random access devices can overlap seek operations (which take relatively long periods of time) on one unit of a device with other operations on other units of the same device. To facilitate this, the support code can divide read and write operations into separate seek operations followed by read or write operations. While the seek operation is taking place on one unit, the support code can start another operation on another unit. For this to work however, the user-written driver code must call the Intel-supplied SEEK$COMPLETE procedure when the seek operation is finished. Chapter 5 explains the SEEK$COMPLETE procedure in more detail.

2.2.2.4 Retries

If an I/O operation fails because of a situation that might not exist every time the operation is attempted (controllers usually categorize these as soft errors), the random access support code doesn't return an exception code immediately. Instead, it retries the operation as many times as the user specifies during configuration. This process is automatic and does not require the user-written driver routines to include any special code. All the driver routines must do is set the status code to E$IO (2BH) and the unit status code to IO$SOFT (2) in the IORS.

2.3 TERMINAL DRIVERS

If you use one of the Intel-supplied terminal drivers, or if you write your own driver and adhere to the terminal driver model, you have access to all of the capabilities of the I/O System's Terminal Support Code.

These capabilities include using control characters to control terminal I/O, redefining those control characters, setting connection and terminal modes (including setting up character translation and simulation), using an auto-answer modem, inquiring about the current terminal setup, limiting a terminal to one connection, and programmatically inserting text into the terminal's input stream.

The following paragraphs describe how to use the Terminal Support Code to control a terminal.

2.3.1 Terminal I/O

There are three buffers involved whenever a task reads input from a terminal: the raw input buffer, the Terminal Support Code input buffer, and the application task's buffer. Each terminal device-unit has its own raw input buffer and its own Terminal Support Code input buffer. Each task that reads input from a terminal has its own buffer. Figure 2-1 shows how these buffers interact.

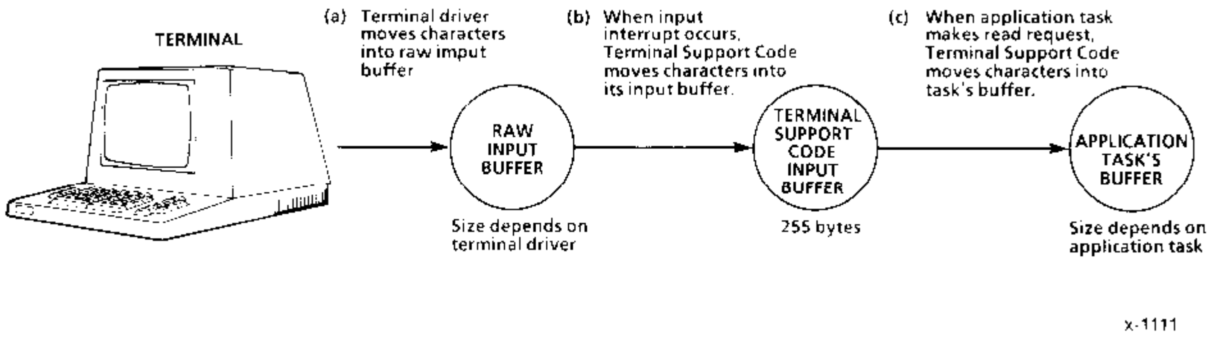

TERMINAL
(a) Terminal driver moves characters into raw impur buffer
(b) When input interrupt occurs, Terminal Support Code moves characters into its input buffer.
(c) When application task makes read request, Terminal Support Code moves characters into task's buffer.
RAW INPUT BUFFER
TERMINAL SUPPORT CODE INPUT BUFFER
APPLICATION TASK'S BUFFER
Size depends on terminal driver
255 bytes
Size depends on application task
x-1111

Figure 2-1. Buffers Used in Terminal I/O

As Figure 2-1 shows, input characters pass through all three buffers on their way from the terminal to the application task.

1. First, the terminal driver takes characters from the device (the terminal) and places them into the raw input buffer.
2. When the device driver signals the Terminal Support Code that an input interrupt has occurred, the Terminal Support Code transfers the characters from the raw input buffer to the Terminal Support Code input buffer.

3. When the I/O System passes a read request to the Terminal Support Code (because a task called A$READ or S$READ$MOVE and specified a connection to a terminal), the Terminal Support Code moves the characters from its input buffer to the task buffer for this read request (the task specified a pointer to this buffer in its A$READ or S$READ$MOVE call).

2.3.1.1 Raw Input Buffer

The size of the raw input buffer, and where the buffer resides, depends on the type of terminal driver. For nonbuffered terminal devices (devices that do not have dual-port memory of their own), the terminal driver must create a logical segment for the raw input buffer when it initializes the unit. For buffered terminal devices, the raw input buffer resides in the dual-port memory of the terminal controller board. Refer to the "Terminal Initialization Procedure" section of Chapter 6 to see how the raw input buffer is initialized.

When the Terminal Support Code transfers characters from the raw input buffer to its input buffer, the number of characters in the raw input buffer depends on the type of terminal device.

Buffered terminal devices do not need to send an interrupt each time an input character is transmitted, so there might be many characters in the raw input buffer when an input interrupt occurs. The maximum number depends on the size of the buffered device's input buffer for that device.

Nonbuffered devices must send one interrupt for each input character, so there is usually only one character in the raw input buffer at a time. However, the raw input buffer enables other input characters to be sent while the Terminal Support Code is processing the previous input character. For nonbuffered devices, the size of the raw input buffer provided by Intel-supplied drivers is 256 bytes.

2.3.1.2 Terminal Support Code Input Buffer

The size of the Terminal Support Code's input buffer is fixed. The buffer for each terminal device-unit is 256 decimal bytes long. Each Terminal Support Code input buffer is divided into two logical buffers: a type-ahead buffer and a line-edit buffer. How input characters move through these logical buffers and into the application task's buffer depends on the terminal's input mode.

If the terminal's input mode is line-edit mode, characters first move into the type-ahead buffer. They move from the type-ahead buffer to the line-edit buffer when the user performs line-editing functions (the characters used to perform line-editing functions are described later in the "Line-Editing Functions" section of this chapter). When the Terminal Support Code receives a read request from the I/O System, it moves the line-edited characters to the requesting application task's buffer.

The maximum number of characters that an application task can request of a terminal in line-edit mode is 253 decimal. (This allows one character for a line terminator, usually a carriage return or a line feed.) If the terminal operator tries to type more than 253 characters before typing a line terminator, the Terminal Support Code discards each extra character and echoes a bell (CONTROL-G) to the terminal.

If a terminal's input mode is transparent or flush mode, the line-edit buffer is not used. Characters move from the type-ahead buffer to the application task's buffer without being line-edited. However, even though the characters aren't line-edited, the Terminal Support Code still might modify some of the characters before placing them into the application task's buffer. Output control characters, OSC sequences, and Terminal Character Sequences can all be intercepted and modified by the Terminal Support Code, depending on the terminal's current connection modes. (Later sections in this chapter describe these character sequences and the connection modes used to enable or disable them.) If you want to ensure that all types of characters can be received without modification when the terminal is in transparent or flush mode, you must also set the output control mode and the OSC control mode so that the Terminal Support Code does not act on these characters when they appear in the input stream.

2.3.1.3 Difference between Transparent and Flush Mode

The difference between transparent mode and flush mode is how the Terminal Support Code treats read requests.

In flush mode, the read request returns immediately with as many characters as currently reside in the Terminal Support Code's input buffer, up to the number of characters requested. This means that any number of characters, from 0 to the number requested, might move into the application task's buffer.

In transparent mode, the read request does not return until all the characters requested by the application task are moved into the application task's buffer.

The maximum number of characters that can be read in one request, in either transparent or flush mode, is 255 for nonbuffered devices and 255 plus the size of the device's dual-port memory for buffered devices.

CAUTION

In transparent mode, if any characters become lost during transmission, an input request can remain unsatisfied. In this case, the terminal will appear to be nonfunctional.

2.3.2 Controlling Terminal I/O

The Terminal Support Code supplies a set of control functions that, when placed in the input stream of data, affect the manner in which data flows between the Basic I/O System and a terminal. There are two kinds of control functions: line-editing functions and output functions. The control characters assigned to these functions are configurable (refer to the "Control Character Redefinition" section of this chapter), but a default set of control characters is provided. The next two sections discuss these control characters.

Not all the characters described in the next two sections take effect when entered from a terminal running under the Human Interface CLI. The only control functions that still operate under the CLI are the delete character, line terminator character, empty type-ahead buffer character, start output character, and stop output character. Refer to the *Operator's Guide to the Extended iRMX II Human Interface* for information concerning the special characters that are available with the CLI.

In these two sections, the term "current line" refers to the set of characters (possibly with editing having been performed on them) that the operator has entered since the most recently entered line terminator.

2.3.2.1 Line-Editing Functions

The control functions that the Terminal Support Code uses to edit data in the line-edit buffer are described in the next few paragraphs, along with the default control characters assigned to perform those functions. Each control character described here can be replaced with a different control character by means of control character redefinition, which is described later in this chapter.

NOTE

The line-editing control characters described in the following paragraphs are effective only when the terminal's mode is line-edit mode (the default mode) and when the characters appear in the input stream. The characters have no effect when the terminal is in transparent or flush mode, or when the characters appear in the output stream.

Function	Default Control Character(s)	Description
Line terminator	Carriage Return or Line Feed	Terminates the current line. Entering either a carriage return or a line feed causes the Terminal Support Code to insert a carriage return and a line feed into the current line. After receiving a line terminator, the Terminal Support Code moves the current line (or the number of characters specified in the input request, if the request is for fewer characters than are in the current line) from the type-ahead buffer, through the line-edit buffer (if line-editing mode is in effect), to the task's buffer. If characters remain in the line-edit buffer, the Terminal Support Code uses them to satisfy the next request for input from the terminal.
Delete character	Rubout	Removes the last data character from the current line. That is, the rubout character and the data character immediately preceding the rubout character in the current line are both removed from the current line. If the terminal has a display screen, the character combination (backspace)(space)(backspace) is echoed to the screen. If the terminal output is hard copy, the deleted character is displayed a second time, surrounded by "#" characters. For example, the sequence "CAT(rubout)(rubout)(rubout)" would appear as CAT#TAC# and would enter and remove the letters C, A, and T from the current line.
Quote next character	CONTROL-P	Causes the next character entered to be treated as data, even if that character is normally a line-editing control character. (Output control characters, such as CONTROL-S and CONTROL-Q, perform their normal functions even if preceded by a CONTROL-P.) During line-edit mode, the Terminal Support Code removes the CONTROL-P from the current line but leaves the disabled character that follows in the input stream. Neither the CONTROL-P nor the character that immediately follows it are displayed at the terminal.
Redisplay line	CONTROL-R	Displays a "#" and then skips to the next line and displays the current line with editing already performed. This enables the terminal operator to see the effects of the editing characters entered since the most recent line terminator. If the current line is empty, CONTROL-R displays the previous line. Moreover, if an operator enters CONTROL-R several times successively, the Terminal Support Code displays previous lines (skipping those that consist of carriage return/line feed only) until it can't find any more lines; then it repeatedly displays the last line found for the remaining CONTROL-R's.
Empty type-ahead buffer	CONTROL-U	Immediately empties the type-ahead buffer the Terminal Support Code manages.
Delete line	CONTROL-X	Deletes the current line. CONTROL-X discards all characters entered since the most recent line terminator and causes "#" to be displayed.
End of file	CONTROL-Z	Terminates the current line (used to signify the end of file). CONTROL-Z differs from Carriage Return and Line Feed in that CONTROL-Z does not become part of the current line. Consequently, entering CONTROL-Z causes a task pending on an A$READ call to have its read request satisfied without transferring the end-of-file character to the waiting task's buffer. If this character is the only character on a line, no characters will be sent in response to the read request.
Special line terminator	None	Terminates the current line without inserting a carriage return/line feed into the text stream. The Terminal Support Code transfers this special line terminator to the waiting task's buffer, but it does not expand the line terminator into a carriage return/line feed pair.

2.3.2.2 Controlling Output to a Terminal

When sending output to a terminal, the Terminal Support Code always operates in one of four modes. The current output mode can be switched dynamically to any of the other output modes by entering an output control character at the terminal. The output modes and their characteristics are as follows:

Normal	The Terminal Support Code accepts output from tasks and immediately passes the output to the terminal for display. This is the default mode.
Stopped	The Terminal Support Code accepts output from tasks (limited by the size of the output buffer), but it queues the output rather than immediately passing it to the terminal.
Scrolling	The Terminal Support Code accepts output from tasks (limited by the size of the output buffer), and it queues the output as in the stopped mode. However, rather than completely preventing output from reaching the terminal, it sends a predetermined number (called the scrolling count) of lines to the terminal whenever an operator enters an appropriate output control character at the terminal.
Discarding	The Terminal Support Code discards all output for the terminal, rather than queuing it or passing it to the terminal.

The following output control characters, when entered at the terminal, change the output mode for the terminal. Like the input control characters, each control character described here is the default, and each can be replaced with a different control character by means of control character redefinition (explained later in this chapter).

NOTE

The output control characters described in the following paragraphs perform their intended operations only when they appear in the input stream. They have no effect when they appear in the output stream.

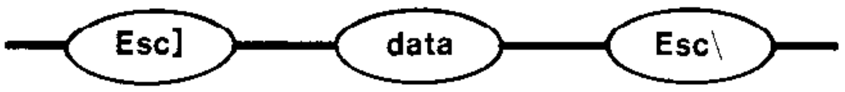
Function	Default Control Character(s)	Description
Discard output	CONTROL-O	Places output into or out of discarding mode. If the output is not in discarding mode, CONTROL-O places output into discarding mode. On the other hand, if output is in discarding mode, CONTROL-O places output into the mode it was in prior to entering discarding mode.
Start output	CONTROL-Q	Places output into normal mode. However, if the last output control character was CONTROL-S, the output mode returns to what it was before entering stopped mode. Note that this implies the following: • The CONTROL-S, CONTROL-Q sequence always returns the output mode to what it was before the sequence was begun. • The CONTROL-Q, CONTROL-Q sequence always places output into normal mode.
Stop output	CONTROL-S	Places output into stopped mode. However, if output was in the discarding mode, CONTROL-S leaves it in discarding mode, but a subsequent CONTROL-O places it in stopped mode.
Scroll one line	CONTROL-T	Places output into scrolling mode, temporarily sets the scroll count to one, sends one output line to the terminal, and places output into stopped mode.
Scroll n lines	CONTROL-W	Places output into scrolling mode and sends n lines to the terminal (where n is the current scrolling count), and places output into stopped mode.

2.3.3 Software Control Strings

The Terminal Support Code enables you to set terminal modes and inquire as to their current setting. The mechanism you use for setting or inquiring is the Software Control String, as defined in the American National Standards Institute publication ANSI X3.64 (1979). There are two kinds of Software Control Strings: the Operating System Command (OSC) sequence and the Application Program Command (APC) sequence.

2.3.3.1 OSC Sequence

The Operating System Command sequence, or OSC sequence, is used by a program or a terminal operator to communicate with the Operating System via the Terminal Support Code. You can use OSC sequences to set the mode of the terminal or to request information about the current modes. The format of the OSC sequence is as follows:



1822

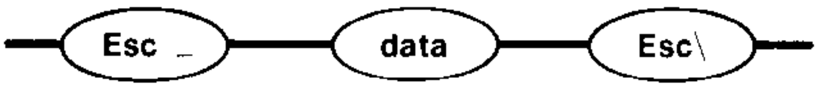
The opening delimiter (Escape Right Bracket) informs the Terminal Support Code the data that follows is an OSC sequence, and the closing delimiter (Escape Backslash) indicates the end of the sequence.

The Terminal Support Code can be set up to accept OSC sequences as input from the terminal operator, as output from a task (via A$WRITE or S$WRITE$MOVE, for example), from both, or from neither. The "Connection Modes" section of this chapter describes how to set up the Terminal Support Code to accept OSC sequences.

When the Terminal Support Code is set up to accept OSC sequences, it strips the OSC sequence from the input or output stream and performs the desired operation.

2.3.3.2 APC Sequence

The Application Program Command sequence (or APC sequence), is used by the Operating System (in this case, the Terminal Support Code) to send information to an application program or terminal. For example, if you use an OSC sequence to request information about the current mode of your terminal, the Terminal Support Code responds by sending an APC sequence containing the requested information. The format of the APC sequence is as follows:



1823

The opening delimiter (Escape Underline) informs the application program (or the operator) the data that follows is an APC sequence, and the closing delimiter (Escape Backslash) indicates the end of the sequence.

When sending an APC sequence, the Terminal Support Code inserts the characters that make up the sequence into the terminal's input buffer, just as if the operator had typed them. This allows the application program to read the characters. The APC sequence is echoed at the terminal if echo mode is enabled (a later section describes enabling and disabling echo mode).

2.3.4 Modes of Terminal Operation

A terminal supported by the Terminal Support Code is governed by numerous modes of operation. Some of these modes apply directly to the terminal, and are independent of the connection a task uses to communicate with the terminal. The remaining modes depend entirely upon the connection being used. The following sections discuss these modes.

A terminal operator or a program can set these modes by issuing OSC sequences. Figure 2-2 shows an overall syntax diagram of the possible OSC sequences. The remainder of this chapter discusses portions of the diagram in more detail. When reading the remainder of this chapter, remember you can combine individual portions of OSC sequences as shown in Figure 2-2.

Instead of using OSC sequences, your programs can use the A$SPECIAL or S$SPECIAL system call to set most of the modes described in this chapter. Those that A$SPECIAL cannot set are noted when described.

When a terminal is attached (during system initialization or when logging onto the system), its default terminal and connection modes are those that were assigned during system configuration. The values specified in the Unit Information Screen associated with the terminal's DUIB are used as the default modes.

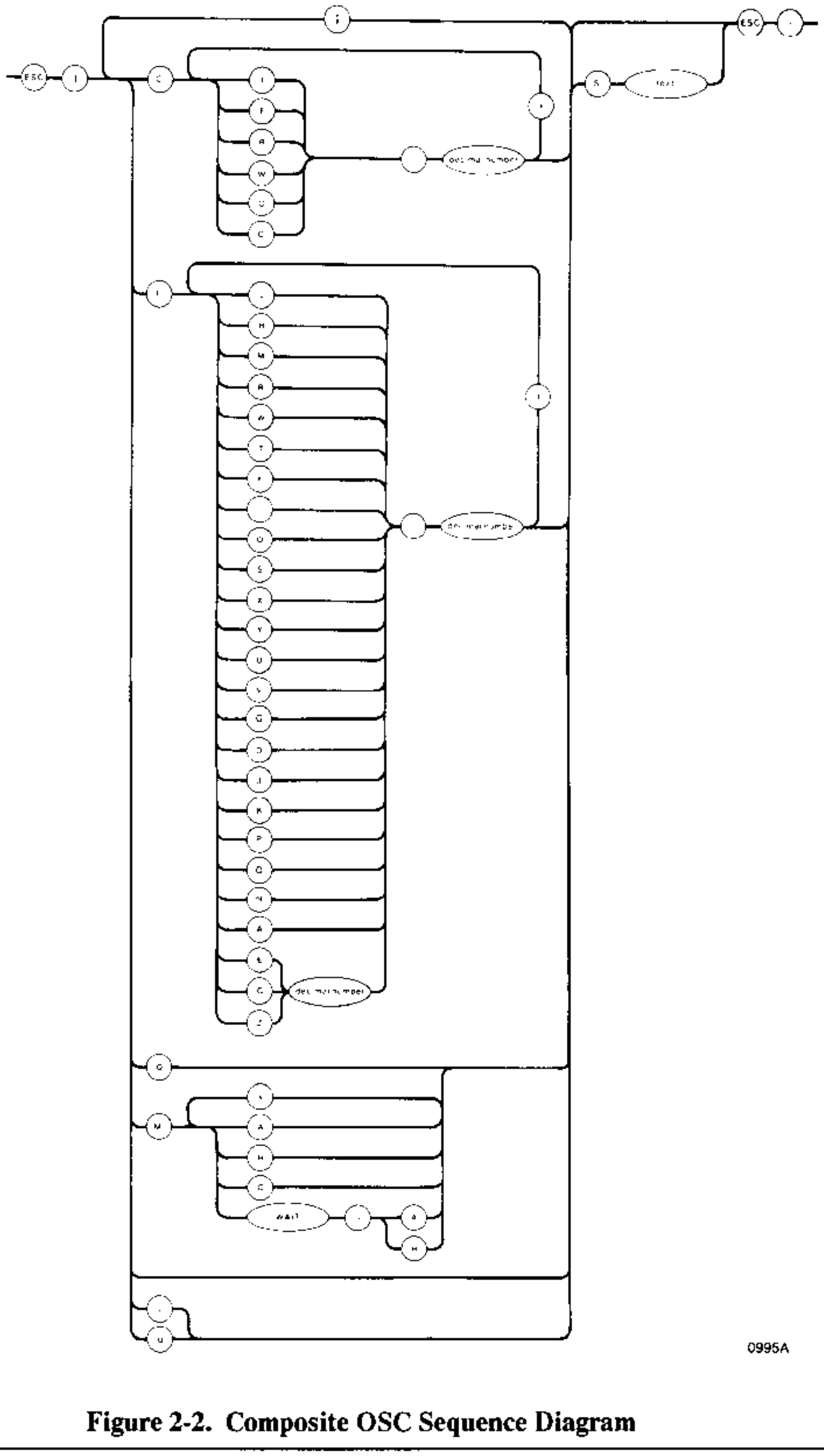


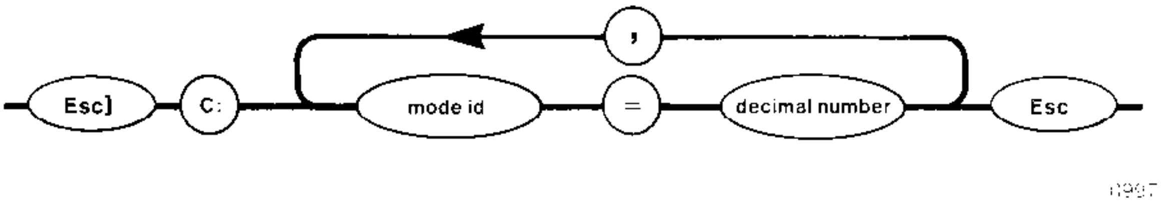
Figure 2-2. Composite OSC Sequence Diagram

2.3.4.1 Connection Modes

This section describes the modes that depend on the connection to the terminal, rather than on the terminal itself. With these modes, when multiple connections to a terminal exist, the terminal might operate one way when communicating via the first connection and a different way when communicating via the second connection.

Each of these modes relates directly to one or more bits in the connection$flags word for the connection (as defined in the *Extended iRMX II Basic I/O System Calls* manual description of the A$SPECIAL system call). The names of the modes, the single-letter identification codes for the modes, the bits of the connection$flags word to which they correspond, and a brief description of their functions are given in Table 2-1.

Assuming the OSC control mode is set appropriately, the modes a terminal inherits from a connection can be altered. The syntax of an OSC sequence that will change one or more of these modes is as follows:



where

C:	Indicates this sequence applies to a connection. You must include the colon (:) after the C.
mode id	An ID letter from the list of modes given in Table 2-1.
decimal number	The value to which you want to set the mode. This number must be of the character data type.

Table 2-1 contains a brief description of the modes and values. For a more complete description, refer to the description of A$SPECIAL in the *Extended iRMX II Basic I/O System Calls* manual.

Table 2-1. Connection Modes

Mode Name	ID	Bit(s)	Description and Values	
Input	T	0-1	0 =	Invalid entry. This value is reserved for future use.
			1 =	Transparent mode. Input is transmitted to the requesting task without being line-edited. Before being transmitted, data accumulates in a buffer until the number of characters equals the number requested by the task in its read call.
			2 -	Line editing mode. Input remains in the line-edit buffer until a line terminator is entered. While in the line-edit buffer, input control characters can be used to edit the input. In line-editing mode, the Terminal Support Code restricts input lines to 254 characters (plus a line terminator, such as carriage return or line feed). If an operator enters more than 254 characters before a task makes an input request, only the first 254 are passed to the requesting task's buffer. The remaining characters are lost. If there are more characters than requested in the buffer when a line terminator is entered, only the requested characters are sent. The additional characters remain in the buffer for the next input request.
			3 =	Flush mode. Input is transmitted to the requesting task without being line-edited. Before being transmitted, data accumulates in a buffer until an input request occurs (that is, a task issues a read request). Then, the number of characters requested is moved from the Terminal Support Code input buffer to the requesting task's buffer. If characters remain in the buffer, they are saved for the next input request. If not enough characters are in the buffer, the request is returned immediately with all available characters, without waiting for the number of characters requested.
Echo	E	2	0 =	The Terminal Support Code echoes characters to the terminal's display screen.
			1 =	No echoing.
Input parity setting	R	3	0 =	For characters entered at the terminal, the Terminal Support Code sets the parity bit to 0.
			1 =	The Terminal Support Code does not alter the input parity bit.
Output parity setting	W	4	0 =	For characters sent to the terminal, the Terminal Support Code sets the parity bit to zero.
			1 =	The Terminal Support Code does not alter the output parity bit.
Output control	O	5	0 =	The Terminal Support Code recognizes and acts on output control characters in the input stream.
			1 =	The Terminal Support Code ignores output control characters in the input stream.
OSC control	C	6-7	0 =	The Terminal Support Code recognizes and acts on OSC sequences that appear in either the input or output stream.
			1 =	The Terminal Support Code acts on OSC sequences in the input stream only.
			2 =	The Terminal Support Code acts on OSC sequences in the output stream only.
			3 =	The Terminal Support Code does not act on any OSC sequences.

NOTE

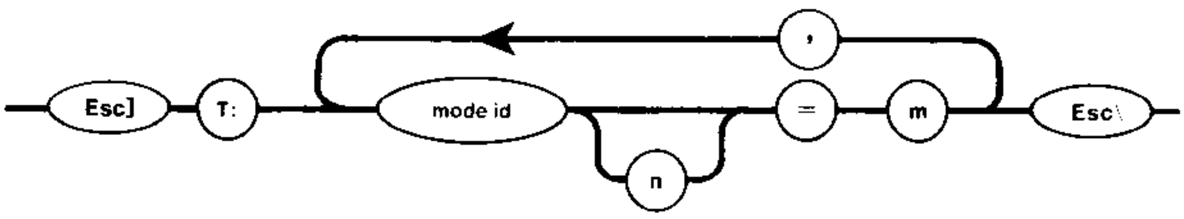
It is possible to use two or more connections concurrently to obtain input from a single terminal. In such cases, the connection associated with the last active read request always has its connection modes in effect. This means that if characters come in from the terminal before another connection's read request has been issued to receive those characters, the characters are processed in the Terminal Support Code's input buffer according to the connection modes associated with the previous read request. To prevent data loss or corruption of data when using connections with different connection mode settings, ensure that read requests occur before data comes in from the terminal.

2.3.4.2 Terminal Modes

In addition to the modes a terminal inherits from a connection, a terminal has modes that are the same regardless of the connection used to communicate with it. This section describes these terminal modes.

Most of the terminal modes relate directly to information supplied as input to the A$SPECIAL system call, either as one or more bits in the terminal$flags word for the connection, or as other words in the terminal$attributes structure (refer to the description of the A$SPECIAL system call in the *Extended iRMX II Basic I/O System Calls* manual for more information about this structure). However, some of the modes have no relation to A$SPECIAL. The names of the modes, the single-letter identification codes for the modes, the bits of the terminal$flags word to which they correspond (if applicable), and a brief description of their functions are given in Table 2-2. The modes that do not correspond to options in A$SPECIAL are noted with asterisks (*) in Table 2-2.

Assuming that the OSC control mode is set appropriately (see Table 2-1), a terminal's modes can be altered using OSC sequences. The syntax of an OSC sequence that changes one or more of the modes covered in this section is as follows:



0998

where

T:	Indicates this sequence applies to a terminal. You must include the colon (:) after the T.
mode id	An ID letter from the list of modes given in Table 2-2.
n	This parameter is valid only if the mode ID is C, E, or Z. It is the decimal representation of an ASCII code (if the mode ID is C or Z) or the number of an escape sequence listed in Table 2-3 (if the mode ID is E).
m	If the mode ID is C, this parameter represents a function code from Table 2-6. If the mode ID is M, it is the number of a terminal character sequence listed in Table 2-4. If the mode ID is Z, it is an integer from 0 to 3 that specifies the index into the special character array. Otherwise it is the value to which you want to change the mode, as listed in Table 2-2.

Table 2-2. Terminal Modes (continued)

Mode Name	ID	Bit(s)	Description and Values
Line protocol	L	1	0 = Full duplex. 1 = Half duplex.
Output medium	H	2	0 = Video display terminal. 1 = Printed (hard copy).
Modem indicator	M	3	0 = No modem connected. 1 = The terminal is connected to the hardware by a modem.
Input parity handling	R	4-5	For drivers that support link parameters, the physical link mode (ID N) when enabled overrides this setting. (Bit 15 of the physical link field enables and disables that mode.) 0 = Driver always sets input parity bit to 0. This yields 8-bit data. 1 = Driver never alters the input parity bit. This yields 8-bit data. 2 = Driver expects even parity on input. This yields 7-bit data. 3 = Driver expects odd parity on input. This yields 7-bit data. Except for the Terminal Communications Controller driver, if an error occurs when even or odd parity is set, the driver sets the eighth bit to one. Errors include (a) a parity error, (b) the received stop bit has a value of 0 (framing error), or (c) the previous character received has not yet been fully processed (overrun error). For the Terminal Communications Controller driver, if a parity error occurs, the character is discarded. If a framing error occurs, the character is returned as an 8-bit null character (00H) without error indication.
Output parity handling	W	6-8	For drivers that support link parameters, the physical link mode (ID N) when enabled overrides this setting. 0 = Driver always sets output parity bit to 0. This yields 8-bit data. 1 = Driver always sets the output parity bit to 1. This yields 8-bit data. 2 = Driver sets output parity bit to give even parity. This yields 7-bit data. 3 = Driver sets output parity bit to give odd parity. This yields 7-bit data. 4 = Driver does not change parity. This yields 8-bit data. **NOTE** If you set input or output parity to even or odd, you must set both of them to the same value. That is, if you set mode ID R to 2 or 3, you must also set mode ID W to the same value.

Table 2-2. Terminal Modes (continued)

Mode Name	ID	Bit(s)	Description and Values
Translation	T	9	Indicates whether the Terminal Support Code for this terminal performs translation between ANSI Standard X3.64 escape sequences and unique terminal character sequences. 0 = Do not enable translation. 1 = Enable translation.
Axes sequence and orientation	F	10-12	Each bit in this three-bit field corresponds to a different function. Enter a value (0-7) accordingly. Bit 10--terminal axis sequence 0 = List or enter the horizontal coordinate first. 1 = List or enter the vertical coordinate first. Bit 11--horizontal axis orientation 0 = Numbering of coordinates increases from left to right. 1 = Numbering of coordinates decreases from left to right. Bit 12--vertical axis orientation 0 = Numbering of coordinates increases from top to bottom. 1 = Numbering of coordinates decreases from top to bottom.
Input baud rate	I	N/A	Corresponds to in$baud$rate field of terminal$attributes structure in A$SPECIAL. 0 = Not applicable. 1 = Perform an automatic baud-rate search. other = Actual input baud rate, such as 2400.
Output baud rate	O	N/A	Corresponds to out$baud$rate field of terminal$attributes structure in A$SPECIAL. 0 = Not applicable. 1 = Use the input baud rate for output. other = Actual output baud rate, such as 9600.
Scrolling number	S	N/A	Corresponds to scroll$lines field of terminal$attributes structure in A$SPECIAL. Specify the number of lines of output to send to the terminal's display whenever the operator enters the scrolling control character (default is CONTROL-W).
Screen width	X	N/A	Corresponds to low-order byte of x$y$size field in A$SPECIAL's terminal$attributes structure. This is the number of character positions on each line of the terminal's screen.
Screen Height	Y	N/A	Corresponds to high-order byte of x$y$size field in A$SPECIAL's terminal$attributes structure. This is the number of lines on the terminal's screen.

Table 2-2. Terminal Modes (continued)

Mode Name	ID	Bit(s)	Description and Values
Cursor addressing offset	U	N/A	Corresponds to low-order byte of x$y$offset field in A$SPECIAL's terminal$attributes structure. This value starts the numbering sequence on both axes.
Overflow offset	V	N/A	Corresponds to high-order byte of x$y$offset field in A$SPECIAL's terminal$attributes structure. This is the value to which the numbering of the axes must "fall back" after reaching 127.
Flow control	G	0	Corresponds to flow control bit in special$modes field of terminal$attributes structure in A$SPECIAL. This bit specifies whether an intelligent communications board (such as an iSBC 544A, iSBC 188/48, iSBC 186/410, or iSBC 188/56 board) sends flow control characters to prevent input buffer overflow. 0 = Disable flow control. 1 = Enable flow control.
Special character	D	1	Corresponds to special character bit of special$modes field of terminal$attributes structure in A$SPECIAL. If your device supports special characters (currently, only the iSBC 188/48, iSBC 188/56, iSBC 186/410, iSBC 546, iSBC 547, and iSBC 548 boards do), the device can send an interrupt whenever a special character (defined later in the special array) is typed. When Special Character Mode is on, the device uses interrupts to inform the Terminal Support Code that special characters have been entered. If a special character has also been defined as a signal character (refer to the description of A$SPECIAL in the *Extended iRMX II Basic I/O System Calls* manual), the Terminal Support Code sends a unit to the appropriate signal semaphore as soon as it receives the special character interrupt. This enables the special character to be processed ahead of characters in the input buffer that are waiting to be processed. However, the special character remains in the input stream and must also be processed in line with the rest of the input characters. If the special character is not assigned as a signal character, the Terminal Support Code discards the special character after receiving it. When Special Character mode is off, the device sends special characters through the normal input stream. The setting of this bit is as follows: 0 = Disable Special Character Mode. 1 = Enable Special Character Mode. The Special Character High Water mark (A) is used in conjunction with this field to control Special Character Mode.

Table 2-2. Terminal Modes (continued)

Mode Name	ID	Bit(s)	Description and Values
High water mark	J	N/A	Corresponds to high$water$mark field of terminal$attributes structure in A$SPECIAL. This field specifies the number of bytes the terminal communication board's buffer must contain before the board sends the flow control character to stop input. (The iSBC 544A board always uses a value of 20 for the high water mark. It ignores this field.)
Low water mark	K	N/A	Corresponds to low$water$mark field of terminal$attributes structure in A$SPECIAL. This field specifies the number of bytes the terminal communication board's buffer must drop to before the board sends the flow control character to start input. (The iSBC 544A board always uses a value of 20 for the low water mark. It ignores this field.)
Start input character	P	N/A	Corresponds to fc$on$char field of terminal$attributes structure in A$SPECIAL. This decimal value specifies an ASCII character that the communication board sends when the buffer drops to the low water mark. (The iSBC 544A board always sends an XON character.)
Stop input character	Q	N/A	Corresponds to fc$off$char field of terminal$attributes structure in A$SPECIAL. This decimal value specifies an ASCII character that the communication board sends when the buffer rises to the high water mark. (The iSBC 544A board always sends an XOFF character.)
Physical link	N	N/A	Corresponds to link$parameter field of terminal$attributes structure in A$SPECIAL. This field specifies characteristics of the physical link between the terminal and a device. It is not supported by all device drivers. When enabled for a supported driver, this control mode overrides the input and output parity modes (IDs R and W). The value in the two low-order bits (0 and 1) specifies the input and output parity, as follows: 0 = No parity 1 = Invalid value 2 = Even parity 3 = Odd parity The value in the next two bits (2 and 3) specifies the character length, as follows: 0 = 6 bits/character 1 = 7 bits/character 2 = 8 bits/character 3 = Invalid value

Table 2-2. Terminal Modes (continued)

Mode Name	ID	Bit(s)	Description and Values
			The value in the next two bits (4 and 5) indicates the number of stop bits, as follows: 0 = 1 stop bit 1 = 1-1/2 stop bits 2 = 2 stop bits 3 = Invalid value Bits 6 through 14 are reserved and should be set to zero. Bit 15 specifies whether the physical link is enabled or disabled. Setting this bit to 1 enables the physical link. Setting the bit to 0 disables the feature. For the Terminal Communications Controller driver, if a parity error occurs on input, the character is discarded. If a framing error occurs, the character is returned as an 8-bit null character (00H). This method of error reporting is different than the method used when the TERMINAL$FLAGS parity specification is in effect.
Special high water mark	A	N/A	Corresponds to spc$hi$water$mark field in terminal$attributes structure of A$SPECIAL. This field is used in conjunction with the Special Characters field (D) to control Special Character Mode. When the device's input buffer fills to contain the number of characters specified in this field, Special Character Mode is enabled (assuming the Special Character field is turned on). If the number of characters in the device's input buffer is less than the high water mark, Special Character Mode is disabled, even if the Special Character field is turned on. If the Special Character field (D) is turned off, this field has no effect.
* Control characters	C	N/A	Modifies the line-edit character and output control character assignments. Refer to the "Control Character Redefinition" section for more information.
* Escape sequence	E	N/A	Pairs an escape sequence with a terminal character sequence to translate or simulate a terminal function. Tables 2-3 and 2-4 list the values you can enter. Refer to the "Translation and Simulation" section of this chapter for more information.

* Corresponds to an option not available via A$SPECIAL. The OSC Query sequence does not return information about this option.

Table 2-2. Terminal Modes

Mode Name	ID	Bit(s)	Description and Values
Special Array	Z	N/A	Corresponds to special$char array of terminal$attributes field in A$SPECIAL. This array can hold as many as four characters that are defined as the device's special characters. If Special Character Mode is on (and the device supports Special Character Mode), typing any of these characters at the keyboard generates an interrupt that immediately informs the Terminal Support Code that a special character has been entered. If the character is a signal character, the Terminal Support Code processes it immediately. If the character isn't a signal character, the Terminal Support Code does nothing with the character. The format of this sequence is Zn = m, where **m** is an integer in the range 0-3, specifying this character's index in the special character array. **n** is a decimal value of the special character's ASCII code. If you define less than four special characters, you must fill the remaining slots of the array with duplicates of the last character you define.

TRANSLATION AND SIMULATION. The translation and simulation capabilities of the Terminal Support Code enable application programs to perform some terminal functions, such as direct control of a terminal's cursor, setting tabs, and other functions, by using a table of predefined escape sequences. This section describes these capabilities.

Intelligent terminals recognize certain ASCII codes (usually control characters or escape codes) as instructions to perform terminal functions. This section refers to these codes as terminal character sequences. Unfortunately, the function performed by a particular terminal character sequence varies from terminal to terminal. For example, one type of terminal might interpret a CONTROL-P as a Cursor Right; a second type of terminal might interpret the same CONTROL-P as a Cursor Down. As a result, an application program that uses terminal character sequences to manipulate terminals must be modified whenever the program is used with a different type of terminal.

The Terminal Support Code recognizes certain escape sequences (a sequence of characters beginning with the Escape character) as instructions to perform terminal functions. Table 2-3 lists the escape sequences the Terminal Support Code supports. These escape sequences remain the same regardless of the terminal you connect to your system. To make your application programs terminal-independent, you can use escape sequences to control your terminal.

The Terminal Support Code can translate these device-independent escape sequences into device-dependent terminal character sequences. How this translation occurs depends on an OSC sequence supplied either by an operator or by an application program. This OSC sequence forms an association between a terminal character sequence and an escape sequence. If translation for the terminal is turned on, the Terminal Support Code replaces the escape sequence with the equivalent terminal character sequence. If translation for the terminal is turned off, or if no association has been formed between the escape sequence and a terminal character sequence, the Terminal Support Code passes the escape sequence unchanged to the terminal.

The Terminal Support Code can also translate a single escape sequence into multiple terminal character sequences. This operation is useful for simulating operations that the terminal doesn't support directly.

NOTE

> The Terminal Support Code translates escape sequences into terminal character sequences consisting of a single control character or an Escape followed by a single character. If your terminal requires sequences that are more complicated, or that require characters other than Escape as the first character in the sequence, you cannot use the Terminal Support Code for your translation. Your tasks must send the other sequences directly. Table 2-4 lists the terminal character sequences that the Terminal Support Code supports.

The concept of translation and simulation centers on the interrelation of three items: terminal character sequence, escape sequence, and OSC sequence.

Terminal Character Sequence

A sequence of characters that is terminal-dependent. It is usually a control character or an escape code. Table 2-4 lists the terminal character sequences that the Terminal Support Code supports.

For example, for a Hazeltine 1510 terminal, CONTROL-H (ASCII hex code 8) is a terminal character sequence that means Cursor Left. Esc followed by CONTROL-E (ASCII hex code 1B 5) is another terminal character sequence that means read cursor address, and so forth. For a different type of terminal, the same functions might require a different set of terminal character sequences.

Although all the terminal character sequences of the Hazeltine 1510 are supported by the Terminal Support Code, some terminals have sequences that are not supported. For example, a Zentec 30 terminal uses the sequence Esc G 2 (ASCII hex code 1B 47 32) to indicate blinking mode. This terminal character sequence is not listed in Table 2-4, and therefore it is not supported.

Escape Sequence	A terminal-independent sequence of characters beginning with an Esc character. Table 2-3 defines the ANSI Standard X3.64 escape sequences that the Terminal Support Code recognizes. Each escape sequence corresponds to a terminal function. If translation is turned on, whenever the escape sequence is sent to the terminal, the Terminal Support Code replaces it with the functionally equivalent terminal character sequence. Alternatively, the Terminal Support Code can either pass the escape sequence to the terminal as is, or it can discard the sequence.
OSC Sequence	A sequence of characters sent to the Terminal Support Code to establish a pairing between an escape sequence and a terminal character sequence. As other sections in this chapter explain, OSC sequences can also set other attributes of the terminal and the connection. To send an OSC sequence, an operator can place the OSC sequence in a file and copy the file to :CO:, or a task can call A$WRITE (or S$WRITE$MOVE) to send the OSC sequence to the terminal. (The operator cannot enter the sequence directly from the terminal.) The Terminal Support Code intercepts the OSC sequence and establishes the desired pairing, regardless of whether the OSC sequence comes from a file or a task.

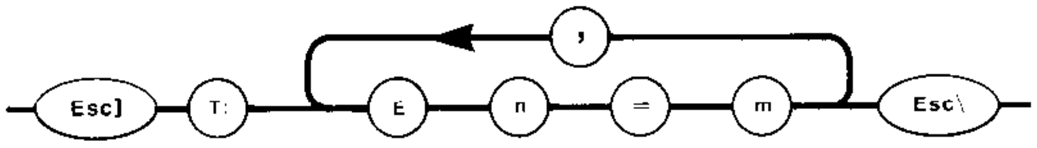
PREPARING THE TERMINAL SUPPORT CODE. OSC sequences can be placed in a file and copied to the terminal, or they can be issued from a task. To establish a pairing, the following conditions must exist:

- There must be a connection to the terminal, and it must be open for writing.
- The OSC control bits for that connection must be set to permit the Terminal Support Code to recognize and act upon OSC sequences on output. This feature can be configured into the system with the ICU, or a task can use the A$SPECIAL or S$SPECIAL system calls to enable the I/O System to act on OSC sequences on output.

When these conditions exist, the operator can copy a file containing OSC sequences to the terminal, or a task can call A$WRITE to send the OSC sequences to the terminal.

Regardless of whether the OSC sequences came from a task or from copying a file to the terminal, the Terminal Support Code intercepts the OSC sequence, removes it from the input or output stream, and establishes the desired pairing.

SYNTAX. The syntax of an OSC sequence that establishes one or more escape-sequence/terminal-character-sequence pairings is as follows:



0999

where

T:	Indicates that this sequence applies to the terminal. You must include the colon (:) after the T.
E	Indicates that this sequence applies to Escape sequences.
n	The number of an escape sequence listed in Table 2-3.
m	The number of a terminal character sequence listed in Table 2-4.

For example, suppose a terminal interprets CONTROL-H (m = 8 in Table 2-4) as a terminal character sequence that causes the cursor to move backward one position. Table 2-3 shows that the Terminal Support Code uses the escape sequence "Esc [D" (n = 3) to mean the same thing. To establish a relationship between m = 8 for the terminal and n = 3 for the Terminal Support Code, the operator or a task can send the following OSC sequence:

```
Esc ] T: E3=8 Esc\
```

Then, if translation is turned on for the terminal (Esc] T: T = 1 Esc\), whenever a task writes the escape sequence "Esc [D" to the terminal, the terminal's cursor will move backward one position. Figure 2-3 illustrates this situation.

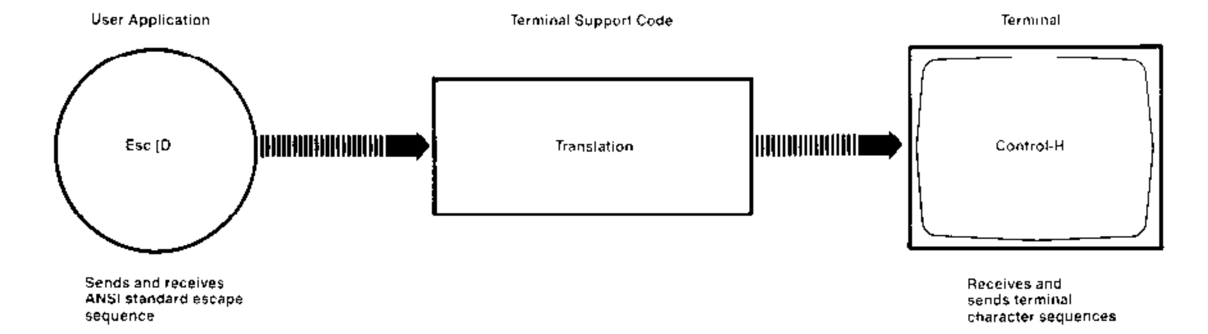

User Application
Terminal Support Code
Terminal
Esc [D
Translation
Control-H
Sends and receives ANSI standard escape sequence
Receives and sends terminal character sequences

Figure 2-3. Escape Sequence Translation

2268

TRANSLATION. If translation is turned on for a terminal, translation occurs when a task calls A$WRITE to write an escape sequence. Instead of simply passing to the terminal an escape sequence the terminal doesn't recognize, the Terminal Support Code intercepts the escape sequence and sends the equivalent terminal character sequence in its place. This equivalence is established by OSC sequences.

Translation also occurs when a task calls A$READ to read a terminal character sequence for which an equivalent escape sequence is established.

Before translation can occur, the operator or the task must turn on translation for the terminal by sending the following OSC sequence:

```
Esc ] T: T=1 Esc\
```

Table 2-2 describes all the terminal modes that can be changed.

If translation is turned off, the Terminal Support Code does not intercept escape sequences. Instead, it passes them on unchanged to the terminal. Changing the "T=1" to "T=0" in the previous OSC sequence turns off translation mode.

TRANSLATION EXAMPLES. This section lists several translation examples for Hazeltine 1510 terminals. All numbers are in decimal unless specified as hexadecimal. These examples assume the terminal's switches are set to allow the Esc character (rather than the tilde character) as the lead-in character of the terminal character sequence. The Terminal Support Code cannot handle terminal character sequences that begin with the tilde character. These examples also assume the following OSC sequence has been issued to specify information about the terminal's coordinate system:

```
Esc ] T:
```

F=0,	(Horizontal coordinates listed first, horizontal numbering increases left to right, vertical numbering increases top to bottom)

`U=96,`	(Axis numbering starts at 96)
`V=32,`	(Axis numbering falls back to 32 after reaching 127)
`X=80,`	(Screen width is 80 characters)
`Y=24,`	(Screen height is 24 lines)
`E6=49,`	(Cursor-addressing terminal character sequence is Esc CONTROL-Q)
`E31=47,`	(Terminal character sequence to clear a line is Esc CONTROL-O)
`E26=51`	(Terminal character sequence to delete a line is Esc CONTROL-S)

```
Esc\
```

The "Cursor Positioning" section of this chapter provides more information about setting up the terminal's coordinate system.

Example 1. Move the cursor to the position X=2, Y=2.

Escape sequence (task)	Terminal Character Sequence (terminal)
`Esc [ 2 ; 2 H` (ASCII Hex Code 1B 5B 32 3B 32 48)	`Esc CONTROL-Q a a` (ASCII Hex Code 1B 11 61 61)

Example 2. Clear the current line from the cursor position to the end of the line.

Escape sequence (task)	Terminal Character Sequence (terminal)
`Esc [ 0 K` (ASCII Hex Code 1B 5B 30 4B)	`Esc CONTROL-O` (ASCII Hex Code 1B 0F)

Example 3. Delete a line.

<u>Escape sequence (task)</u>   <u>Terminal Character Sequence (terminal)</u>

```
Esc [ 1 M                      Esc CONTROL-S
(ASCII Hex Code 1B 5B          (ASCII Hex Code 1B 13)
31 4D)
```

SIMULATION. Simulation occurs when there is no single terminal character sequence corresponding exactly to a given escape sequence. Simulation is necessary because some terminals might not have terminal character sequences to perform the functions indicated by certain escape sequences. Therefore, the Terminal Support Code must simulate that function.

Simulation is performed only on output. That is, a task can call A$WRITE and simulation will occur. Simulation does not occur when the task calls A$READ.

When a task calls A$WRITE to write an escape sequence corresponding to a simulated function, the Terminal Support Code intercepts the escape sequence and figures out what the task wants the terminal to do. Then the Terminal Support Code sends a series of one or more terminal character sequences that the terminal does recognize, producing the desired effect. Figure 2-4 illustrates this concept.

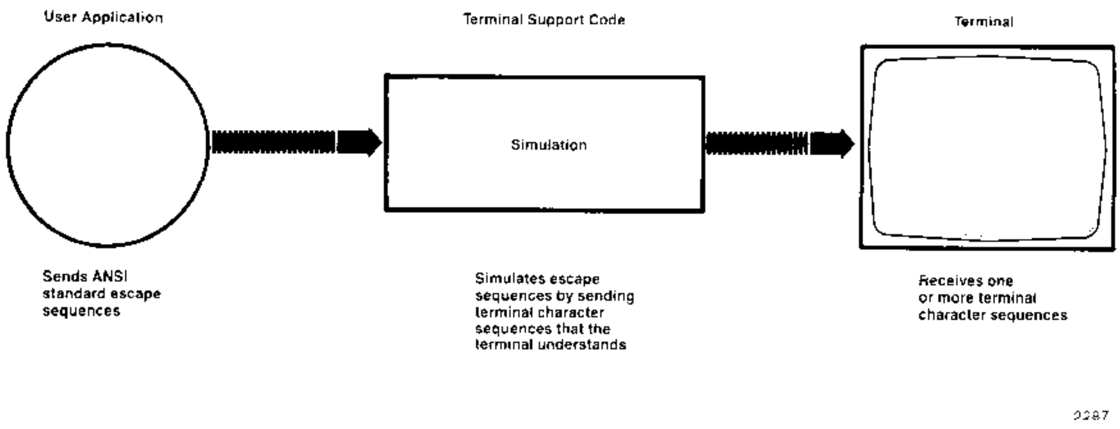

User Application
Terminal Support Code
Terminal
Simulation
Sends ANSI standard escape sequences
Simulates escape sequences by sending terminal character sequences that the terminal understands
Receives one or more terminal character sequences
0287

Figure 2-4. Escape Sequence Simulation

For example, suppose the terminal does not support tab stops. If given the right information about the terminal, the Terminal Support Code can simulate the tab stops, creating the impression the terminal does indeed support tab stops as if it were a typewriter. All the Terminal Support Code must do to accomplish this is to

- Remember where the cursor is on the display.
- Remember where the tab stops are supposed to be.
- Be able to tell the terminal to move the cursor forward by one space.

In general, to support simulation of escape sequences, the terminal must have terminal character sequences for the following cursor movements:

- One position to the right
- One position to the left
- One position upward
- One position downward

The Terminal Support Code cannot simulate all the escape sequences listed in Table 2-3. It can simulate only the sequences numbered 0, 1, 6-11, 13, 15, 18-20, 22, and 23.

SIMULATION EXAMPLES. This section lists three simulation examples for a hypothetical terminal (all numbers are in decimal unless specified as hexadecimal). These examples assume the terminal has the following terminal character sequences for cursor movement:

Cursor Movement	Terminal Character Sequence
Cursor up	CONTROL-L (ASCII Hex code 0C)
Cursor down	CONTROL-K (ASCII Hex code 0B)
Cursor left	CONTROL-H (ASCII Hex code 08)
Cursor right	CONTROL-J (ASCII Hex code 0A)

In addition, the examples assume the following OSC sequence has been sent to translate the right, left, up, and down cursor movements:

```
Esc ] T: E2=10, E3=8, E4=12, E5=11 Esc\
```

Example 1. Move the cursor to x=2, y=8 (assuming the current position is x=1, y=5).

The escape sequences will then be simulated as follows:

Escape Sequence (Output from Task)	Terminal Character Sequence (Actually Sent to Terminal)
Esc [8 ; 2 H	CONTROL-J CONTROL-K CONTROL-K CONTROL-K
(ASCII Hex code 1B 5B 38 3B 32 48)	(ASCII Hex code 0A 0B 0B 0B)

Example 2. Simulate tab stops.

Although the terminal does have a terminal character sequence for moving to the right (m = 10 in Table 2-4, which corresponds to escape sequence n = 2 in Table 2-3), it does not support functions n = 10 (advancing to the next tab stop) and n = 11 (setting a tab stop) listed in Table 2-3. Therefore, the Terminal Support Code must simulate these functions. The following OSC sequence sets up the terminal to support tabs:

```
        Esc ] T:E2=10, E3=8, E4=12, E5=11, E10=192, E11=192 Esc\
```

Before operators can set tab stops, they must provide the Terminal Support Code with the location of the cursor. This can be done by resetting the terminal; that is, by sending the following escape sequence to the terminal:

```
        Esc c          (n=0 in Table 2-3)
```

Resetting the terminal works only if the terminal has a reset terminal command and if the operator has established a relationship between that command and the escape sequence Esc c using an OSC sequence (Esc] T:E0 = m Esc\, where m is the number of a terminal character sequence listed in Table 2-4).

Having done this, you can set a horizontal tab stop by entering Esc [0 W at the terminal, and you can advance the cursor to the next tab stop by entering Esc [1 I. The Terminal Support Code keeps track of the locations of the horizontal tab stops as well as the position of the cursor.

TABLE OF ESCAPE SEQUENCES. Table 2-3 lists the escape sequences you can pair with terminal character sequences by means of OSC sequences. The following remarks apply to Table 2-3:

- The "Code" column contains codes used in the ANSI X3.64 document.
- The expression "99" represents any decimal number. Unless otherwise specified, omitting the number causes the Terminal Support Code to supply a default value of 1.

- In some cases, you can combine multiple escape sequences into a single, compound escape sequence. The table identifies these cases.
- The Terminal Support Code can simulate the escape sequences numbered 0, 1, 6 through 11, 13, 15, 18 through 20, 22, and 23. The remaining escape sequences can only be translated.
- In almost all cases, tasks issue the escape sequences by calling A$WRITE. The exceptions concern escape sequences 7 and 18, and they are described in the table.

Table 2-3. Escape Sequence (continued)

n	Code	Escape Sequence	Function
† 0	RIS	Esc c	Returns the terminal to its initial state. This consists of resetting the horizontal tab stops to four spaces apart, beginning with the first space, and returning the cursor to the upper-left corner of the display.
† 1	HTS	Esc H	Sets a horizontal tab at the current cursor position.
2	CUF	Esc [99 C	Moves the cursor forward the specified number of positions.
3	CUB	Esc [99 D	Moves the cursor backward the specified number of positions.
4	CUU	Esc [99 A	Moves the cursor upward the specified number of positions.
5	CUD	Esc [99 B	Moves the cursor downward the specified number of positions.
† 6	CUP	Esc [99 ; 99 H	Moves the cursor to the position specified by the decimal numbers. The first number specifies the vertical coordinate position, and the second number specifies the horizontal coordinate position. The horizontal coordinates are numbered from left to right, beginning with 1, and the vertical coordinates are numbered from top to bottom, also beginning with 1. If the parameters are omitted, this sequence moves the cursor to the upper-left corner of the display.
† 7	CPR	Esc [99 ; 99 R	Reports the coordinates of the current cursor position. The Terminal Support Code places this sequence into the terminal's input stream in response to sequence number 19, which asks for the cursor's coordinates. The first number specifies the vertical coordinate position, and the second number specifies the horizontal coordinate position. The horizontal coordinates are numbered from left to right, beginning with 1, and the vertical coordinates are numbered from top to bottom, also beginning with 1.
† 8	CBT	Esc [99 Z	Moves the cursor backward by the specified number of horizontal tab stops. For example, if the specified number is 2, the cursor moves backward to the second tab stop it encounters.
† 9	CHA	Esc [99 G	Moves the cursor to the specified position in the current line.
† 10	CHT	Esc [99 I	Moves the cursor forward by the specified number of horizontal tab stops. For example, if the specified number is 2, the cursor moves forward to the second tab stop that it encounters.
† 11	CTC	Esc [0 W	Sets a horizontal tab stop at the current cursor position. You can combine this and any other CTC escape sequence to form a compound CTC escape sequence. An example of such a combined sequence is ESC [0;1 W, which sets both horizontal and vertical tab stops at the cursor position.

† Function that can be simulated.

Table 2-3. Escape Sequences (continued)

n	Code	Escape Sequence	Function
12	CTC	Esc [1 W	Sets a vertical tab stop at the current cursor position. See the description of escape sequence number 11.
† 13	CTC	Esc [2 W	Clears a horizontal tab stop if there is one at the current cursor position. See the description of escape sequence number 11.
14	CTC	Esc [3 W	Clears a vertical tab stop if there is one at the current cursor position. See the description of escape sequence number 11.
† 15	CTC	Esc [4 W	Clears all horizontal tab stops on the line containing the cursor. See the description of escape sequence number 11.
16	CTC	Esc [5 W	Clears all horizontal and vertical tab stops. See the description of escape sequence number 11.
17	CTC	Esc [6 W	Clears all vertical tab stops. See the description of escape sequence number 11.
† 18	DA	Esc [99 c	Tasks send this sequence with the number 0 to request the ID number of the terminal to which the request is being sent. The Terminal Support Code intercepts the request and returns to the requesting task an identical sequence, except that the number (which is greater than 0) is the requested ID number.
19	DSR	Esc [6 n	Asks the Terminal Support Code to report the coordinates of the current cursor position. See sequence number 7 for a description of the response.
† 20	TBC	Esc [0 g	Clears a horizontal tab stop if there is one at the current cursor position. You can combine this and any other TBC escape sequence to form a compound TBC escape sequence. An example of such a combined sequence is ESC [0;1 g, which clears both horizontal and vertical tab stops from the current cursor position.
21	TBC	Esc [1 g	Clears a vertical tab stop if there is one at the current cursor position. See the description of escape sequence number 20.
† 22	TBC	Esc [2 g	Clears all horizontal tab stops on the line containing the cursor. See the description of escape sequence number 20.
† 23	TBC	Esc [3 g	Clears all horizontal and vertical tab stops. See the description of escape sequence number 20.
24	TBC	Esc [4 g	Clears all vertical tab stops. See the description of escape sequence number 20.
25	DCH	Esc [99 P	Deletes the specified number of characters, beginning at the current cursor location.

† Function that can be simulated.

Table 2-3. Escape Sequences (continued)

n	Code	Escape Sequence	Function
26	DL	Esc [99 M	Deletes the specified number of lines, beginning at the line containing the cursor.
27	ECH	Esc [99 X	Replaces the specified number of characters with blanks, beginning at the current cursor location.
28	ED	Esc [0 J	Places blanks at all positions from the cursor to the end of the display. You can combine this and any other ED escape sequence to form a compound ED escape sequence. An example of such a combined sequence is ESC [0;1 J, which clears the entire display.
29	ED	Esc [1 J	Places blanks at all positions from the beginning of the display to the cursor. See the description of escape sequence number 28.
30	ED	Esc [2 J	Fills the entire display with blanks. See the description of escape sequence number 28.
31	EL	Esc [0 K	Places blanks at all positions from the cursor to the end of the line. You can combine this and any other EL escape sequence to form a compound EL escape sequence. An example of such a combined sequence is ESC [0;1 K, which places blanks throughout the line currently containing the cursor.
32	EL	Esc [1 K	Places blanks at all positions from the beginning of the line containing the cursor to the cursor itself. See the description of escape sequence number 31.
33	EL	Esc [2 K	Places blanks at all positions in the line containing the cursor. See the description of escape sequence number 31.
34	ICH	Esc [99 @	Inserts the specified number of blanks, beginning at the location of the cursor.
35	IL	Esc [99 L	Inserts the specified number of blank lines, beginning at the location of the cursor.
36	NP	Esc [99 U	Moves the display forward in a multiple-page file by the specified number of pages. If the specified number of pages is 0, the display moves to the next page.
37	PP	Esc [99 V	Moves the display backward in a multiple-page file by the specified number of pages. If the specified number of pages is 0, the display moves to the previous page.
38	SD	Esc [99 T	Moves the display downward (forward) by the specified number of lines. If the specified number of lines is 0, the display moves to the next line.
39	SU	Esc [99 S	Moves the display upward (backward) by the specified number of lines. If the specified number of lines is 0, the display moves to the previous line.
40	SGR	Esc [99 m	See the comment following this table.

Table 2-3. Escape Sequences (continued)

n	Code	Escape Sequence	Function
41	RM	Esc [0 l	An error condition.
42	RM	Esc [1 l	See the comment following this table.
43	RM	Esc [2 l	Unlocks the terminal's keyboard, allowing all characters to be entered.*
44	RM	Esc [3 l	Prevents control characters from being displayed, but still causes those characters to have their normal effects.*
45	RM	Esc [4 l	Causes output characters to overwrite characters on the display.*
46	RM	Esc [5 l	See the comment following this table.
47	RM	Esc [6 l	See the comment following this table.
48	RM	Esc [7 l	See the comment following this table.
49	RM	Esc [8 l	Reserved.
50	RM	Esc [9 l	Reserved.
51	RM	Esc [10 l	See the comment following this table.
52	RM	Esc [11 l	See the comment following this table.
53	RM	Esc [12 l	Causes characters to be displayed on the terminal's display screen as they are entered.
54	RM	Esc [13 l	See the comment following this table.
55	RM	Esc [14 l	See the comment following this table.
56	RM	Esc [15 l	See the comment following this table.
57	RM	Esc [16 l	See the comment following this table.
58	RM	Esc [17 l	See the comment following this table.
59	RM	Esc [18 l	Causes horizontal tab stops to apply equally to all lines, rather than on a line-by-line basis.*
60	RM	Esc [19 l	Causes data on the terminal's display screen to be treated as a continuous stream, rather than as a collection of disjoint, independent pages.*
61	RM	Esc [20 l	Prevents the line feed character from automatically performing a carriage return when sent to the terminal.*
62	SM	Esc [0 h	An error condition.

* This is the normal (default) setting for most terminals.

Table 2-3. Escape Sequences

n	Code	Escape Sequence	Function
63	SM	Esc [1 h	See the comment following this table.
64	SM	Esc [2 h	Locks the terminal's keyboard, preventing characters from being received when they are typed.
65	SM	Esc [3 h	Enables the display of control characters for debugging purposes.
66	SM	Esc [4 h	Enables output characters to be inserted in the display, rather than always overwriting existing characters.
67	SM	Esc [5 h	See the comment following this table.
68	SM	Esc [6 h	See the comment following this table.
69	SM	Esc [7 h	See the comment following this table.
70	SM	Esc [8 h	Reserved.
71	SM	Esc [9 h	Reserved.
72	SM	Esc [10 h	See the comment following this table.
73	SM	Esc [11 h	See the comment following this table.
74	SM	Esc [12 h	Prevents characters from being displayed on the terminal's screen as they are typed.
75	SM	Esc [13 h	See the comment following this table.
76	SM	Esc [14 h	See the comment following this table.
77	SM	Esc [15 h	See the comment following this table.
78	SM	Esc [16 h	See the comment following this table.
79	SM	Esc [17 h	See the comment following this table.
80	SM	Esc [18 h	Causes horizontal tab stops to apply only to the line on which they are entered.
81	SM	Esc [19 h	Causes data to be treated as a collection of disjoint, independent pages. In this kind of environment, a terminal operator typically accesses the various pages in a file by pressing keys such as "next page", "previous page", or "go to page".
82	SM	Esc [20 h	Causes the line feed character to automatically perform a carriage return when sent to the terminal.

Comment: This mode (or sequence or function) is included for completeness, but a description is beyond the scope of this manual. For details, refer to the 1979 version of the ANSI X3.64 standard.

TABLE OF TERMINAL CHARACTER SEQUENCES. Table 2-4 lists the terminal character sequences that you can pair with escape sequences via OSC sequences. The value "m" in the table is the decimal representation of the code the terminal requires for the given function. As the table shows, if the function requires a character plus a lead-in Escape, add 32 to the character's decimal representation. Note that the ASCII code 1BH (Escape) by itself cannot be the result of a translation.

Recall the assignment portion of the proper OSC sequence has the form En = m, where n is the escape sequence number and m is the terminal character sequence number.

Table 2-4. Terminal Character Sequences

m	Terminal Character Sequence or Special Instructions
0	Disable the translation of escape sequence n. That is, pass the escape sequence through to the terminal without Terminal Support Code translation or simulation.
1	01H (CONTROL-A)
2	02H (CONTROL-B)
.	
.	
.	
26	1AH (CONTROL-Z)
27	This terminal character sequence (1BH - Escape) is not supported.
28	1CH (FS)
29	1DH (GS)
30	1EH (RS)
31	1FH (US)
32	Esc 00H
33	Esc 01H
.	
.	
.	
159	Esc 7FH
160-191	Reserved
192	Simulate the escape sequence.
193	Discard the escape sequence. That is, do not translate or simulate it, and do not pass it on to the terminal.

CURSOR POSITIONING. Before the Terminal Support Code can monitor or control the position of a cursor, it must know the coordinate numbering conventions for that terminal. The Terminal Support Code has its own "model" of the terminal coordinate numbering scheme. As mentioned in Table 2-3, this model is the following:

- The horizontal coordinates are numbered from left to right, beginning with 1.
- The vertical coordinates are numbered from top to bottom, also beginning with 1.

Whenever programs refer to cursor positions, they should use this convention.

Although this seems a reasonable way to refer to positions on a terminal screen, not all terminals use this numbering scheme. But, the Terminal Support Code can translate the terminal numbering scheme into its own model, as long as the terminal numbering scheme obeys the following rules:

- The numbering of the axes can start at any point left or right, top or bottom. However, the numbering of both axes must start with the same positive value.
- From there, numbering of both axes must increase by ones until (or unless) it reaches 127.
- If the numbering of an axis reaches 127, it must then "fall back" to a lower positive value; whereafter, it must again increase by ones.
- If the numbering of both axes reaches 127, the numbering of each must fall back to the same value.

If the terminal numbering scheme meets these criteria, you can set up the Terminal Support Code (via OSC sequences) to handle that numbering scheme. The terminal modes F, U, V, X, and Y (as listed in Table 2-2) enable you to specify information about the terminal numbering conventions. Once you send the proper OSC sequences, the Terminal Support Code translates the terminal numbering conventions into its own standard conventions. Then, your programs can use the Terminal Support Code standard conventions when referring to all terminals.

For example, suppose the terminal horizontal positions (that is, its columns) are numbered left to right as 80, 81, 82, ..., 127, 16, 17, 18, ..., 31. Also, suppose its vertical positions (its rows) are numbered top to bottom as 103, 102, 101, ..., 80. Finally, suppose that when referring to a particular position on the terminal screen, you must specify the vertical position first, followed by the horizontal position. Note that this numbering convention differs from the Terminal Support Code numbering conventions in the following ways:

- The numbering on each axis starts with 80, rather than starting with 1.
- The numbering of the horizontal axis, when it reaches 127, drops back to 16 before resuming its climb.

- The numbering of the vertical axis increases from bottom to top, rather than increasing from top to bottom.
- The coordinates of a given screen position are vertical coordinate first, then horizontal coordinate, rather than being horizontal first and vertical second.

Although the numbering convention of this terminal is unorthodox, it does obey the rules listed earlier in this section. To set up this terminal for use with the Terminal Support Code, you can issue the following OSC sequence:

```
Esc ] T: F=5, U=80, V=16, X=64, Y=24 Esc\
```

The F=5 portion tells the Terminal Support Code the vertical coordinate is called out first, the horizontal numbering increases from left to right, and the vertical numbering increases from bottom to top. The U=80 portion specifies the starting number, V=16 indicates the "fall-back" value, X=64 specifies the line length, and Y=24 specifies the number of lines on the screen. Refer to Table 2-2 for more information about these modes.

Table 2-5 lists OSC sequences you can use to set up the cursor positioning and control characters of some common terminals. The OSC sequences listed in the table do not take full advantages of the features of the terminals, but if you connect the terminals to an Intel integrated microsystem, such as the System 310, these OSC sequences enable you to run the RSAT system analysis test. You can add to these sequences to support more features of the terminals.

Table 2-5. Example OSC Sequences for Common Terminals

Hazeltine 1500, 1510, 1520, Executive 80*	TeleVideo 950*	Description
Esc]	Esc]	(OSC sequence opening delimiter)
T:T=1,	T:T=1,	(Turn on translation)
F=0,	F=1,	(Specify coordinate system of terminal)
U=96,	U=32,	(Starting value of numbering sequence of both axes)
V=32,	V=32,	(Fall back value when cursor position reaches 127 on either axis)
X=80,	X=80,	(Number of character positions per line)
Y=24,	Y=24,	(Number of lines per screen)
E2=16,	E2=12,	(Cursor right)
E3=8,	E3=08,	(Cursor left)
E4=44,	E4=11,	(Cursor up)
E5=43,	E5=22,	(Cursor down)
E6=49,	E6=93,	(Cursor position)
E31=47	E31=148	(Clear line, cursor to end)
Esc\	Esc\	(OSC sequence closing delimiter)

* Hazeltine and Executive 80 are trademarks of Hazeltine Corporation. TeleVideo is a trademark of TeleVideo Systems, Inc.

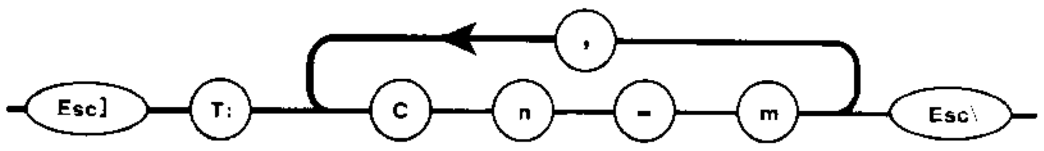
CONTROL CHARACTER REDEFINITION. An earlier section of this chapter listed the default characters assigned to the line-editing and output control functions. The character assignments for these control functions are not fixed. You can dynamically assign any control character to a control function provided by the Terminal Support Code, as described in this section.

If you assign a control character to a control function, the assignment applies only when the character appears as input from the terminal. In particular, assigning a new control character to be the Escape character does not change the Escape character used for output translation. It is still the ASCII ESC character, hexadecimal code 1BH. Also, any new Escape character you define cannot be used as part of an OSC sequence.

The characters you can assign to control functions include the following:

Character	Decimal ASCII Code	Hexadecimal ASCII Code
CONTROL-@	0	0
CONTROL-A through CONTROL-Z	1 - 26	1 - 1AH
ESC	27	1BH
FS	28	1CH
GS	29	1DH
RS	30	1EH
US	31	1FH
DEL	127	7FH

The syntax of the OSC sequence used to assign control characters to control functions is as follows:



0996

where

T: Indicates that this sequence applies to the terminal. You must include the colon (:) at the end.

C Indicates that this sequence applies to control characters.

n The decimal representation of the ASCII code for the desired control character. This decimal value can be in the range 0-31 or 127.

If this control character is already assigned as a signal character, this assignment to a control function is ignored. (Refer to the description of A$SPECIAL in the *Extended iRMX II Basic I/O System Calls* manual for information on assigning signal characters.)

If this control character is assigned to another control function, this OSC sequence reassigns the character to a new function.

m	A number indicating the function to assign to the control character. Table 2-6 lists these numbers, with their corresponding descriptions and defaults.

For example, the following sequence cancels the default assignment of Rubout (DEL) as the deletion character and assigns Backspace (BS) in its place:

```
Esc ] T: C127=0, C8=11 Esc\
```

Table 2-6. Menu of Control Character Functions

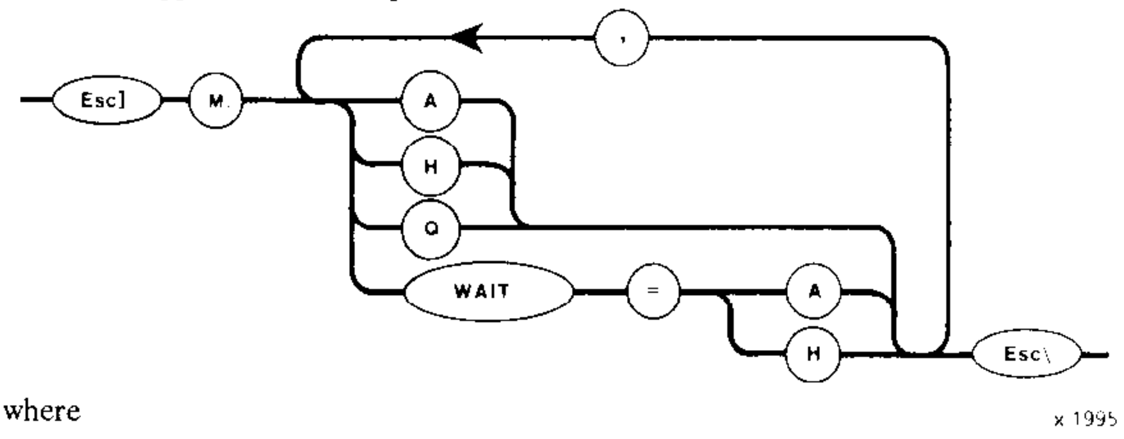
Function Number	Abbreviated Description	Default Assignment
0	Pass character through Unchanged	All control characters not assigned as line- edit, escape, output control, or signal characters
1	Stop output	CONTROL-S
2	Start output	CONTROL-Q
3	Discard output	CONTROL-O
4	Scroll N lines	CONTROL-W
5	Scroll 1 line	CONTROL-T
6	Empty typeahead buffer	CONTROL-U
7	Escape	Escape (ASCII 1BH)
8	Line terminator	CONTROL-J, CONTROL-M
9	End of file	CONTROL-Z
10	Quote next character	CONTROL-P
11	Delete character	Rubout (ASCII 7FH)
12	Delete line	CONTROL-X
13	Redisplay line	CONTROL-R
14	Special line terminator	None

2.3.4.3 Using an Auto-Answer Modem with a Terminal

The Terminal Support Code supports terminals that interface with an Extended iRMX II-based application system through an auto-answer modem. It does this by controlling the RS232 Data Terminal Ready (DTR) line and by providing OSC sequences to enable handshaking between a task and a terminal connected to a modem.

If your system contains a modem, you can configure the Basic I/O System to support modem control (refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual*). Once you do this, the Basic I/O System, during system initialization, establishes the initial link to the modem. Or, your tasks can use OSC sequences to establish modem mode (the M mode listed in Table 2-2), to break the link (hang up), and to reestablish the link (dial and answer). Other than these operations, tasks and terminals communicate through a modem as if linked by a dedicated line.

The following diagram illustrates the syntax of the OSC sequences relating to modem control. Unlike other OSC sequences, only tasks should send these OSC sequences to the Terminal Support Code. An operator at a terminal should never send them.



where

M:	Indicates that this sequence applies to a modem. You must include the colon (:) after the M.
A	Causes the Terminal Support Code to answer the phone (set the DTR line active). This indicates that the task is ready to send or receive data.
H	Causes the Terminal Support Code to hang up the phone (clear the DTR line). This breaks the phone link.
Q	Queries the Terminal Support Code for the status of the modem. In response, the Terminal Support Code sends an APC sequence in this form: `Esc _ M:x Esc\` where x is either "A" if the modem is answered (DTR active) or "H" if the modem is hung up (DTR clear).
WAIT	Requests the Terminal Support Code to notify the task when the modem is in the proper state (only the "W" is required). "W = A" requests notification when DTR becomes active (caused by an operator dialing up the modem). "W = H" requests notification when DTR becomes clear (caused by the operator hanging up the phone). When the modem is in the proper state, the Terminal Support Code inserts an APC sequence of the following form in the input stream: `Esc _ M:x Esc\` where x is either "A" if the modem has been answered (DTR active) or "H" if the modem has been hung up (DTR clear).

The following example illustrates how a task can use the OSC modem sequences to communicate with a terminal via a modem.

Assume that one task is dedicated to monitoring the modem and communicating through it. Assume further that the task has a connection to the modem and that the connection is open for both reading and writing. Typical protocol (using the connection) is the following:

1. The task writes the following OSC sequence to the terminal:

   ```
   Esc ] M:H Esc\
   ```

   This sequence hangs up the phone (breaks the link). It is an initialization step.

2. The task writes the following OSC sequence to the terminal:

   ```
   Esc ] C:T=1,E=1 Esc\
   ```

This sets transparent mode (so the task can later read a certain number of characters or wait until they appear) and turns off echoing to the terminal's screen. These changes are for this connection only, not for other connections (if any) to the modem.

3. The task writes the following OSC sequence to the terminal:

   ```
   Esc ] M:WAIT-A Esc\
   ```

   This requests that the Terminal Support Code return a notification (an APC sequence) when the modem has been answered (DTR becomes active).

4. The task issues a read request to read seven characters from the terminal. Eventually, when DTR becomes active, the Terminal Support Code inserts an APC sequence of the following form in the input stream:

   ```
   Esc_ M:A Esc\
   ```

   This message means a terminal user has dialed up the modem and is ready to communicate.

5. The task writes the following OSC sequence to the terminal:

   ```
   Esc ] M:WAIT=H Esc\
   ```

   This causes the Terminal Support Code to send the APC sequence "Esc_ M:H Esc\" to the task when the terminal user hangs up.

6. The terminal and the task communicate as if on a dedicated line for as long as is necessary. However, whenever the task receives input, it must scan the input for the APC sequence "Esc_ M:H Esc\".

   During this time, the task should operate the modem in transparent or flush mode, rather than line-edit mode. In line-edit mode, each line received from the modem must be terminated with a line terminator (such as a carriage return/line feed). However, the last set of characters (the APC sequence) will probably not be followed by a line terminator. Therefore, if the connection is operating in line-edit mode, the application task will never receive the final hangup message from the Terminal Support Code.

7. Eventually, the operator hangs up the phone. When this happens, the Terminal Support Code inserts the following APC sequence in the input stream:

   ```
   Esc_ M:H Esc\
   ```

   This means the terminal user has hung up and the link is broken.

8. The task returns to step 2.

This protocol is offered as a model and is by no means the only one possible. Note, however, that only the task, and never the terminal, should send OSC sequences to the Terminal Support Code for modem control. This restriction does not apply to other OSC sequences.

Under some circumstances, a task needs to find out whether a terminal is ready to talk to the task via the modem. The task can ascertain the state of the modem (answered or hung up) by performing the following steps, in order:

1. Call A$WRITE to send the following OSC sequence to the modem:

   ```
   Esc ] C:T=1,E=1 Esc\
   ```

   This sets transparent mode (disabling line editing) and turns off the echoing to the terminal's screen. Note that this is for this connection only, not for other connections (if any) to the modem.

2. Call A$WRITE to send the following OSC sequence to the modem:

   ```
   Esc ] M:Q Esc\
   ```

   This requests information as to the status of the modem; that is, answered (A) or hung up (H).

3. Call A$READ to read seven characters from the modem. This receives from the Terminal Support Code an APC sequence of the form:

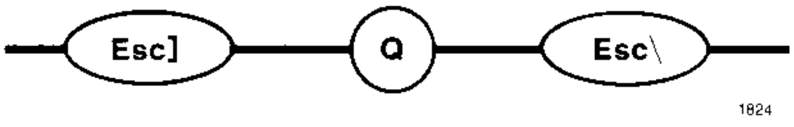
   ```
   Esc_ M:x Esc\
   ```

   where x is "A" if the modem is answered and "H" if the modem is hung up. This technique will work because the Terminal Support Code places the APC sequence, without a line terminator, at the front of the line buffer for the connection where data (if any) is awaiting input requests from the task.

After performing these steps, the task can restore the connection's line editing and echo modes to their original states.

2.3.4.4 Obtaining Information about a Terminal

In addition to specifying information about your terminal, you can use OSC sequences to request information about the terminal's current settings. The syntax of the Terminal Query OSC sequence that requests information about the terminal is as follows:



where

Q	Indicates that this sequence is a query for information.

In response to the Terminal Query OSC sequence, the Terminal Support Code sends an APC sequence that lists the current values of all modes for a terminal and all modes for the connection through which the request was made. However, it does not return information about the escape-sequence/terminal-character-sequence pairings or about the input/output control character assignments.

A task obtains the query information by performing the following steps, in order:

1. Call A$WRITE to send the following OSC sequence to the terminal:

   ```
   Esc ] Q Esc\
   ```

   This queries the Terminal Support Code for information about the terminal. In response, the Terminal Support Code returns the requested information in the form of an APC sequence (without a line terminator) at the front of the typeahead buffer for the connection. If echoing mode is enabled, this information will echo at the terminal when the task reads it.

2. Call A$READ to read the appropriate number of characters from the connection. The number of characters returned depends on the values of the modes, and some of these modes, such as the input baud rate (I) for the terminal, can vary in length. You should allow two spaces for the "Esc_" at the beginning, two spaces for the "Esc\" at the end, and enough spaces for the modes in between. A simple, safe way to obtain this data is to read one byte at a time, until "Esc\" appears. The modes are separated by commas and packed together without blanks. An example of a returned APC sequence follows:

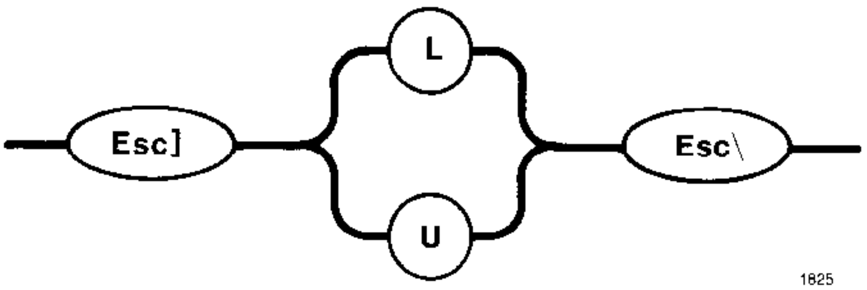
```
Esc_ C:T=2,E=0,R=0,W=1,O=0,C=0;T:L=0,H=0,M=0,R=2,W=2,T=1,F=0,
I=9600,O=0,S=18,X=64,Y=24,U=80,V=16,G=1,J=0,K=0,P=0,Q=0 Esc\
```

2.3.4.5 Restricting the Use of a Terminal to One Connection

If there are multiple connections to a terminal, you can send OSC sequences via any one of the connections to "lock" the terminal. When you do this, the terminal is temporarily restricted from communicating via any other connection.

Tasks that communicate via the first connection can use the connection according to how it was opened, and I/O requests through that connection are processed normally. However, if tasks make I/O requests via the locked-out connections, the Terminal Support Code queues those I/O requests until the terminal is unlocked.

The syntax of the Lock and Unlock OSC sequences are as follows:



where

L	Locks the terminal, temporarily preventing I/O via other connections.
U	Unlocks the terminal, allowing I/O to occur via all connections to the terminal.

The only way to lock a terminal is for a task or a terminal operator to send the Lock OSC sequence. However, there are two ways to unlock a terminal:

- A task (via the connection used to lock the terminal) or the terminal operator can send the Unlock OSC sequence.
- A task can close the connection used to lock the terminal.

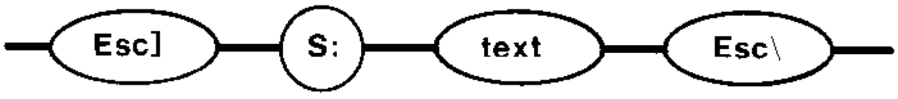
After a terminal is unlocked, the queued I/O requests are processed in the order in which they were queued.

NOTE

If there is a chance of a terminal becoming locked, tasks should use care when using BIOS system calls to communicate via other connections to the terminal. If the tasks invoke system calls such as A$READ and A$WRITE without specifying a response mailbox, a deadlock situation could occur.

2.3.4.6 Programmatically Inserting Data into a Terminal's Input Stream

A task can use an OSC sequence to insert (stuff) data into a terminal's input stream. This process is useful when operators must enter large blocks of data that vary only slightly from one occurrence to the next. The syntax of the Stuffing OSC sequence is as follows:



1826

where

S:	Indicates that this sequence stuffs data into the input stream. You must include the colon (:) after the S.
text	Text (a maximum of 126 characters) to be placed in the terminal's input stream. If the connection's echo mode is enabled, the stuffed text displays on the screen. If the connection's line-editing mode is enabled, the operator can edit the stuffed text.

If you send composite OSC sequences (multiple OSC sequences separated with semicolons), the composite sequence can contain only one Stuffing OSC sequence, and that subsequence must be the last subsequence.

CHAPTER 3
INTEL-SUPPLIED DEVICE DRIVERS

The Operating System supplies a number of complete device drivers that support many different devices. Instead of writing your own device driver, you might be able to use one of the Intel-supplied device drivers to communicate with the devices in your application system.

Some of these Intel-supplied device drivers fall into the random access or common category, some fall into the terminal category, and some are custom drivers. This chapter lists the drivers by category and provides general information about each driver. Table 3-1 lists the drivers included with the Extended iRMX II package.

Table 3-1. Intel-Supplied Device Drivers

Type	Device Driver
Random Access or Common	iSBC 208 flexible disk driver Mass Storage Controller (MSC) driver * iSBX 218A flexible disk driver iSBC 220 SMD driver iSBC 186/224A multi-peripheral driver iSBX 251 bubble memory driver iSBC 264 bubble memory driver Line printer driver for iSBX 350 Line printer driver for iSBC 286/10(A) SCSI driver for iSBC 286/100A
Terminal	iSBC 186/410 terminal driver Terminal Communications Controller driver iSBC 534 terminal driver iSBC 544A terminal driver iSBX 351 terminal driver 8274 terminal driver 82530 terminal driver
Custom	Byte bucket driver RAM driver Stream file driver

* Supports the iSBC 214, iSBC 215G, and iSBX 217C controllers. It also supports the iSBX 218A controller when it is mounted on the iSBC 215G board.

3.1 RANDOM ACCESS AND COMMON DRIVERS

This section describes the random access and common drivers supplied with the Operating System. These drivers are designed according to the guidelines listed in Chapter 5 and they make use of the features supplied by the I/O System's support code for random access and common devices.

For information on adding any of these device drivers to your application system, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

3.1.1 iSBC® 208 Disk Driver

This driver supports flexible diskette drives connected to the iSBC 208 controller. The driver supports up to four units (0-3) per controller.

The iSBC 208 flexible disk driver

- Supports both 8-inch and 5.25-inch disks (single- or double-sided, single- or double-density).
- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, and CLOSE functions.
- Accepts the OPEN function but performs no operations for it.

This driver requires a 1K byte block of memory in the first megabyte to be reserved for use by the driver. You can specify the physical address of this block with the ICU during configuration. The default address is 1600H.

Track formatting and volume change notification are supported via the SPECIAL function.

The seek overlap capabilities of the iSBC 208 controller allow seek operations to occur concurrently on multiple units of the same device. However, concurrent seek operations cannot take place while any other operation is in progress.

Refer to the *iSBC 208 Flexible Disk Drive Controller Hardware Reference Manual* for more information about the iSBC 208 controller.

3.1.2 Mass Storage Controller (MSC) Driver

This driver provides the means to control hard disk drives, flexible disk drives, and tape drives. It can support either the iSBC 215G board or the iSBC 214 board as the base controller.

The iSBC 215G board controls hard disk drives. The iSBX 217C and iSBX 218A boards can be attached to this board, via iSBX connectors, to provide tape and flexible disk support, respectively. The iSBC 214 board provides all of the iSBC 215G, iSBX 217C, and iSBX 218A features in a single board. However, hard disks formatted with either the iSBC 215G or iSBC 214 controller will not work when connected to the other controller.

This driver requires 62 bytes of memory in the first megabyte to be reserved for its use. You can specify the physical address of this block with the ICU during configuration. The default address is 1200H.

3.1.2.1 iSBC® 215G Features

The iSBC 215G controller supports hard disk drives (usually Winchester technology) that are compatible with the ST506/412 interface and connected to the iSBC 215G controller. The driver supports up to four units per controller.

For hard disks, this driver

- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions.
- Accepts the OPEN and CLOSE functions but performs no operations for them.

Track formatting, volume change notification, getting device characteristics, and getting and setting bad track information are supported via the SPECIAL function.

Improves hard disk integrity via the seek-on-detach feature, in which the disk heads seek to the innermost cylinder (which is usually the diagnostic cylinder) in response to the F$DETACH$DEV command.

The seek overlap capabilities of the iSBC 215G controller allow seek operations to occur on multiple units while a DMA transfer is occurring on another Winchester unit.

Refer to the *iSBC 215 Generic Winchester Disk Controller Hardware Reference Manual* for more information about the iSBC 215G controller.

3.1.2.2 iSBX™ 217C Features

The iSBX 217C Magnetic Cartridge Tape Interface Board is an 8-bit, single-wide, iSBX MULTIMODULE I/O expansion board for installation on any 8- or 16-bit iSBC host board that has an iSBX connector. It provides an interface between a MULTIBUS processor board and 1/4-inch magnetic cartridge tape drives that correspond to the QIC-02 interface.

When the iSBX 217C MULTIMODULE board is attached to an iSBC 215G Winchester disk controller board, the driver supports up to four tape drives connected to the MULTIMODULE board, although only one tape drive can be attached at a time. This tape driver

- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, and CLOSE functions.
- Accepts the OPEN function but performs no operations for it.

The driver supports the following subfunctions via the SPECIAL function:

Format
Get device characteristics
Rewind tape
Read tape file mark
Write tape file mark
Retension tape

The only Human Interface commands supported for tape drives are ATTACHDEVICE, DETACHDEVICE, BACKUP, and RESTORE. After you issue a BACKUP or RESTORE command, the software automatically rewinds the tape.

Refer to the *iSBX 217C Magnetic Cartridge Tape Interface Multimodule Board Hardware Reference Manual* for more information about the iSBX 217C MULTIMODULE board.

3.1.2.3 iSBX™ 218A Features

The iSBX 218A board controls flexible disk drives that are compatible with the SA784/460 interface. The driver supports flexible disk drives connected to an iSBX 218A MULTIMODULE board, as long as that MULTIMODULE board is attached to an iSBC 215G Winchester disk controller board. This driver

- Supports 8-inch flexible disks.
- Supports 5.25-inch flexible disks.
- Supports DMA transfers.
- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions.
- Accepts the OPEN and CLOSE functions but performs no operations for them.

The driver supports the following subfunctions via the SPECIAL function:

Formatting tracks
Getting device characteristics
Volume change notification (supported on 5.25-inch flexible disk drives that have a "door-open" signal)

The seek overlap capabilities of the iSBC 215G/iSBX 218A controllers allow seek operations to occur on multiple diskette units, even while a DMA transfer occurs on a Winchester unit. However, all seek operations on diskette units must be complete before a DMA transfer can occur on any of the diskette units.

Refer to the *iSBX 218A Flexible Diskette Controller Board Hardware Reference Manual* for more information about the iSBX 218A controllers.

3.1.2.4 iSBC® 214 Features

The iSBC 214 controller provides the features of the iSBC 215G, iSBX 217C, and iSBX 218A controllers, all in a single board. The driver supports hard disk drives (5.25-inch Winchester technology drives only), 5.25-inch flexible disk drives, and tape drives connected to the iSBC 214 board in the same manner that it supports those devices when they are connected to the iSBC 215G/iSBX 217C/iSBX 218A controller combination.

Improves hard disk integrity via the seek-on-detach feature, in which the disk heads seek to the innermost cylinder (which is usually the diagnostic cylinder) in response to the F$DETACH$DEV command.

Refer to the *iSBC 214 Multi-Peripheral Controller Hardware Reference Manual* for more information about the iSBC 214 board.

3.1.3 iSBX™ 218A Disk Driver

The iSBX 218A flexible disk driver supports up to four flexible disk drives (units 0 through 3) connected to the iSBX 218A MULTIMODULE board, as long as the board is mounted on an intelligent host CPU board that contains an iSBX connector. The iSBX 218A driver

- Supports 5.25-inch flexible disks (single- or double-sided, single- or double-density).
- Does not support DMA transfers (interrupts are disabled during transfers)
- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions.
- Accepts the OPEN and CLOSE functions but performs no operations for them.
- Supports these subfunctions via the SPECIAL function:

  Formatting tracks
  Getting device characteristics
  Volume change notification (supported on 5.25-inch flexible disk drives that have a "door-open" signal)

This driver operates in polled mode. Interrupts are disabled during read, write, and format operations.

Refer to the *iSBX 218A Flexible Diskette Controller Board Hardware Reference Manual* for more information about the iSBX 218A controller.

3.1.4 iSBC® 220 SMD Disk Driver

This driver supports up to four Storage Module Device (SMD) disks connected to the iSBC 220 controller. The driver

- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions.
- Accepts the OPEN and CLOSE functions but performs no operations for them.

This driver requires a block of memory in the first megabyte whose size depends on the number of units configured into the system. The minimum size is 1152 bytes plus the number of bytes equal to the device granularity multiplied by the number of units. You can specify the physical address and size of this block with the ICU during configuration. The default location is 1210H and the default size is 1480H bytes.

Track formatting, volume change notification, and getting device characteristics are supported via the SPECIAL function.

Improves hard disk integrity via the seek-on-detach feature, in which the disk heads seek to the innermost cylinder (which is usually the diagnostic cylinder) in response to the F$DETACH$DEV command.

Refer to the *iSBC 220 SMD Disk Controller Hardware Reference Manual* for more information about the iSBC 220 controller.

3.1.5 iSBC® 186/224 Multi-Peripheral Driver

The iSBC 186/224A multi-peripheral controller is a MULTIBUS II board capable of controlling 12 peripheral devices: four Winchester, four 5.25-inch flexible disk, and four streaming tape drives. Equipped with DIN connectors, the controller provides 16-bit data lines and 32-bit address lines. Two buses, an ADMA-controlled I/O bus and a local bus, enable the controller to communicate with the peripheral devices and the CPU host. The iSBC 186/224A controller resides on the iPSB (parallel system bus).

The iSBC 186/224A device driver is a random access driver that uses the message-passing protocol of the iPSB. The iSBC 186/224A device driver

- Supports up to four Winchester disk drives that are compatible with the ST506 interface and four flexible diskette drives (all units may be accessed simultaneously)
- Supports up to four streaming tape drives (accesses only one tape drive at a time)
- Uses the Extended iRMX II transport protocol for sending and receiving both solicited and unsolicited messages

- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions
- Accepts the OPEN function but performs no operation for it
- Accepts the CLOSE function but performs no operation for it (this function is supported for tape drives only)

Supports the following subfunctions via the SPECIAL function:

Format
Get device characteristics
Rewind tape
Recalibrate disk drive
Read tape file mark (forward searching only, one or more file marks)
Write tape file mark
Get bad track or sector information
Set bad track or sector information
Retension tape

Improves hard disk integrity via the seek-on-detach feature, in which the disk heads seek to the innermost cylinder (which is usually the diagnostic cylinder) in response to the F$DETACH$DEV command.

The only Human Interface commands supported for tape drives are ATTACHDEVICE, DETACHDEVICE, BACKUP, RETENTION, and RESTORE. After you issue a BACKUP or RESTORE command, the software automatically rewinds the tape.

3.1.6 iSBX™ 251 Bubble Memory Driver

The iSBX 251 driver supports the iSBX 251 magnetic bubble memory MULTIMODULE board. The driver

- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions.
- Accepts functions OPEN and CLOSE but performs no operations for them.

Formatting is supported via the SPECIAL function.

Because of the lower performance of the iSBX 251 board, this board does not work reliably with Intel CPU boards that run at clock speeds greater than 6Mhz.

Refer to the *iSBX 251 Magnetic Bubble Memory Multimodule Board Technical Reference Manual* for more information about the iSBX 251 controller.

3.1.7 iSBC® 264 Bubble Memory Driver

The iSBC 264 driver supports the iSBC 264 bubble memory controller board. The driver supports up to four boards per interrupt. Each board can represent a single unit of the device, or a single unit can consist of one to four boards. Each board can have from one to four Intel 7114 bubble devices; however, all boards of a given unit must have the same number of bubble devices. The driver includes these features:

- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions.
- Accepts functions OPEN and CLOSE but performs no operations for them.

Formatting is supported via the SPECIAL function, which performs no function on the iSBC 264 board and immediately returns with the IORS.STATUS field set to E$SPACE.

Refer to the *iSBC 264 Magnetic Bubble Memory Multimodule Board Technical Reference Manual* for more information about the iSBC 264 controller.

3.1.8 iSBX™ 350 Line Printer Driver

The line printer driver is a common driver that provides an interface between the Basic I/O System's physical file driver and the 8255 parallel I/O port of an iSBX 350 MULTIMODULE board. A flat ribbon cable connects the board with a Centronics-type line printer. The line printer driver

- Supports the WRITE, ATTACH$DEVICE, DETACH$DEVICE functions.
- Accepts functions OPEN and CLOSE but performs no operations for them.
- Can be used only with the physical file driver.

3.1.9 iSBC® 286/10(A) Line Printer Driver

This line printer driver is a common driver that provides an interface between the Basic I/O System physical file driver and the 8255 parallel I/O port of the iSBC 286/10, iSBC 286/10(A), and iSBC 286/12 boards. A flat ribbon cable connects this board with a Centronics-type line printer. The line printer driver

- Supports the WRITE, ATTACH$DEVICE, DETACH$DEVICE functions.
- Accepts functions OPEN and CLOSE but performs no operations for them.
- Can be used only with the physical file driver.

3.1.10 SCSI Driver

The SCSI driver supports 5.25- and 8-inch Winchester and diskette controllers that meet the Small Computer System Interface specifications described in the ANSI document ANSI X3T9.2/82-2. It also supports some Shugart Associates System Interface (SASI) controllers. This driver

- Works specifically with the iSBC 286/100A processor board as a host.
- Supports the READ, WRITE, SEEK, SPECIAL, ATTACH$DEVICE, and DETACH$DEVICE functions.
- Accepts the OPEN and CLOSE functions but performs no operations for them.

Track formatting is supported via the SPECIAL function.

3.2 TERMINAL DRIVERS

This section describes the terminal drivers supplied with the Operating System. These drivers are designed according to the guidelines listed in Chapter 6. Unless otherwise noted, the drivers in this section provide all the Terminal Support Code features listed in Chapter 2.

For information on adding any of these device drivers to your application system, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

3.2.1 iSBC® 186/410 Terminal Driver

The iSBC 186/410 terminal driver supports asynchronous terminals connected to the iSBC 186/410 Serial Communications Board. Each iSBC 186/410 controller supports six serial lines.

The iSBC 186/410 driver

- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE functions
- Can be used with only the physical file driver

Getting terminal data, setting terminal data, setting signal characters, setting special characters, setting link parameters, and enabling and disabling flow control are supported via the SPECIAL function.

The iSBC 186/410 controller is a buffered device supporting block input and output of data via solicited and unsolicited messages. The iSBC 186/410 controller supports all Terminal Support Code features including special-character interrupts. The only buffered-device features it does not support are configurable start/stop input characters and high/low-water marks (the controller always uses XON and XOFF for the start and stop characters, and fixed values for high- and low-water marks).

Multiple hosts can share a single iSBC 186/410 controller, but multiple hosts cannot simultaneously share a serial line on the controller.

This driver does not support separate input and output baud rates for a single serial line. The driver recognizes the baud rates 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200. If an unsupported baud rate is requested, the driver overrides it with the next higher supported baud rate.

For more information about the iSBC 186/410 controller, refer to the *iSBC 186/410 Serial Communications Board User's Guide*.

3.2.2 Terminal Communications Controller Driver

The Terminal Communications Controller driver is a terminal driver that supports the following intelligent communications controllers:

iSBC 188/48 controller
iSBC 188/56 controller
iSBC 546 controller
iSBC 547 controller
iSBC 548 controller

All of these controllers are intelligent controllers that can manage buffered input and output. The iSBC 188/48 and iSBC 188/56 controllers can support up to 12 serial communication channels per board. The iSBC 546 controller can support up to four serial communications channels plus one line printer port. The iSBC 547 and iSBC 548 controllers can support up to eight communications channels. The Terminal Communications Controller driver

- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE functions.
- Can be used only with the physical file driver.
- Supports only asynchronous (RS232) communications.

Getting terminal data, setting terminal data, setting signal characters, setting special characters, setting link parameters, enabling and disabling input flow control, setting input flow control start and stop characters, and setting high- and low-water marks are supported via the SPECIAL function.

The controllers supported by this driver are buffered controllers that support all the features of the Terminal Support Code, including special character interrupts.

Although the driver supports baud-rate search and can recognize the baud rates 300, 600, 1200, 2400, 4800, 9600, and 19200, the firmware on the iSBC 188/48 board is set up so that baud-rate search works only if the terminal is set up for seven-bit characters and even parity. At other settings, the baud-rate search fails to recognize the terminal.

The iSBC 188/48 and iSBC 188/56 input buffers are fixed at 1940 bytes. When setting up the flow control parameters, you can specify values in the range of 0-1936 for the low-water mark and values in the range of 8-1936 for the high-water mark. The TCC driver divides the values you supply by eight before passing them to the controller as BYTE values. The controller then multiplies the received value by eight to obtain the actual value. Intel recommends a value of 950 decimal for the low-water mark and 1900 decimal for the high-water mark. Refer to Chapter 2 for more information about flow control.

Refer to the *iSBC 188/48 Advanced Communicating Computer Hardware Reference Manual* for more information about the iSBC 188/48 controller. Refer to the *iSBC 188/56 Advanced Communications Computer Hardware Reference Manual* for more information about the iSBC 188/56 controller. Refer to the *iSBC 546/547/548 High Performance Terminal Controllers Hardware Reference Manual* for more information about the iSBC 546, iSBC 547, and iSBC 548 controllers.

3.2.3 iSBC® 534 Terminal Driver

The iSBC 534 driver is a terminal device driver that supports terminals connected to one or more iSBC 534 Four-Channel Communications Expansion boards, each of which has four USARTs. As many as four iSBC 534 boards can share a single interrupt line, in which case their USARTs are treated as separate units of a single device.

The iSBC 534 driver

- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE functions.
- Can be used only with the physical device driver.

Getting terminal data, setting terminal data, and setting signal characters are supported via the SPECIAL function.

The iSBC 534 board is not a buffered device, so the features of the Terminal Support Code that apply to buffered devices are not supported by the iSBC 534 driver. The only other Terminal Support Code feature that this driver does not support is separate input and output baud rates. The driver supports baud-rate search and can recognize the baud rates 110, 150, 300, 600, 1200, 2400, 9600, and 19200.

Refer to the *iSBC 534 Four Channel Communications Expansion Board Hardware Reference Manual* for more information about the iSBC 534 communications board.

3.2.4 iSBC® 544A Terminal Driver

The iSBC 544A driver is a terminal device driver that supports terminals connected to one or more iSBC 544A Four-Channel Intelligent Communications Expansion boards. Each iSBC 544A board can support four serial lines. As many as four iSBC 544A boards can share a single interrupt line, in which case their channels are treated as separate units of a single device.

The iSBC 544A driver

- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE functions.
- Can be used only with the physical file driver.

Getting terminal data, setting terminal data, setting signal characters, and enabling and disabling flow control are supported via the SPECIAL function.

The iSBC 544A controller is a buffered device that incorporates an 8085A CPU for its on-board processing. Therefore the iSBC 544A driver supports almost all of the features of buffered devices. The only buffered-device features it does not support are special character interrupts, which the iSBC 544A firmware cannot generate, and configurable start/stop input characters and high/low-water marks (the controller always uses XON and XOFF for the start and stop characters, and fixed values for high- and low-water marks).

The only other Terminal Support Code feature that this driver does not support is separate input and output baud rates for a single serial line. The driver supports baud-rate search and is capable of recognizing the baud rates 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200.

Refer to the *iSBC 544A Intelligent Communications Controller Board Hardware Reference Manual* for more information about the iSBC 544A controller.

3.2.5 iSBX™ 351 Terminal Driver

The iSBX 351 Terminal Driver is a terminal driver that supports a terminal device connected to the serial port of an iSBX 351 MULTIMODULE board, an iSBC 386/2X(3X) board, or any board that contains an 8251A USART. The driver requires two interrupt levels, one for input and one for output. The iSBX 351 terminal driver

- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE functions.
- Can be used only with the physical file driver.

Getting terminal data, setting terminal data, and setting signal characters are supported via the SPECIAL function.

The devices supported by this driver are not buffered devices, so the features of the Terminal Support Code that apply to buffered devices do not apply to this driver. The only other Terminal Support Code feature that the iSBX 351 driver does not support is modem control. The driver supports baud-rate search and can recognize the baud rates 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200.

3.2.6 8274 Terminal Driver

The 8274 terminal driver supports a terminal device via either of two on-board serial ports of an iSBC 286/10, iSBC 286/10A, or iSBC 286/12 board. The driver supports a single 8274 MPSC (multi-protocol serial controller) with as many as two terminal devices connected to it. The 8274 MPSC must be configured in nonvectored mode. This means that the 8274 does not send an interrupt type itself, but rather requests an 8259A PIC to interrupt the processor.

The 8274 terminal driver

- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE functions.
- Supports only asynchronous (RS232) communications.
- Can be used only with the physical file driver.

Getting terminal data, setting terminal data, and setting signal characters are supported via the SPECIAL function.

The 8274 MPSC is not a buffered device, so the features of the Terminal Support Code that apply to buffered devices are not supported by the 8274 driver. The driver supports all other Terminal Support Code features except modem control. The driver supports baud-rate search and can recognize the baud rates 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200.

3.2.7 82530 Terminal Driver

The 82530 terminal driver supports terminals via the 82530 Serial Communication Controller of the iSBX 354 MULTIMODULE board or the 82530 SCC mounted on a 286/100A processor board. The 82530 SCCs must be configured in nonvectored mode. Nonvectored mode means that the 82530 does not send an interrupt type itself, but rather requests an 8259A PIC to interrupt the processor.

The 82530 terminal driver

- Supports a single 82530 SCC component containing two serial channels configured as a single device and mounted on either an iSBX 354 MULTIMODULE board or a 286/100A processor board.
- Supports the READ, WRITE, SPECIAL, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE functions.
- Supports only asynchronous (RS232) communications.
- Can be used only with the physical file driver.

Getting terminal data, setting terminal data, and setting signal characters are supported via the SPECIAL function.

The 82530 is not a buffered device, so the features of the Terminal Support Code that apply to buffered devices are not supported by the 82530 driver. The only other Terminal Support Code feature that this driver does not support is separate input and output baud rates for a single serial line. The driver supports baud-rate search and can recognize the baud rates 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200.

3.3 CUSTOM DRIVERS

This section describes the custom drivers supplied with the Operating System. These drivers are designed according to the guidelines listed in Chapter 7. They don't use any of the special features provided for random access, common, or terminal drivers.

For information on adding any of these device drivers to your application system, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

3.3.1 Byte Bucket Driver

This driver provides a pseudo device interface for operations that don't require device activity. It is used for discarding output (byte bucket) and direct communication between tasks (stream files). This driver

- Returns an end of file for any READ operation and write completed for any WRITE operation.
- Accepts all operations except SPECIAL (and SEEK, in the case of stream files) but performs no operations for them.

3.3.2 RAM Driver

The RAM driver allows an area of RAM to be used for temporary storage of named or physical files. This memory is called a RAM-disk. When used in conjunction with the LOCDATA and ADDLOC commands, the RAM-disk can be preloaded with Human Interface commands for immediate access (see the *Operator's Guide to the Extended iRMX II Human Interface* for more information). The RAM driver

- Supports the READ, WRITE, and SPECIAL functions.
- Accepts functions SEEK, ATTACH$DEVICE, DETACH$DEVICE, OPEN, and CLOSE but performs no operations for them.
- Supports up to 16 units.

Because the RAM-disk emulates a disk storage device, you can attach and format the RAM-disk after booting the system, and you can access the RAM-disk as if it were an ordinary disk (until the system is reset).

If you intend to preload the RAM-disk with Human Interface commands, you must use the LOCDATA command to create an object module (which contains an image of an already-formatted RAM-disk) and use the ADDLOC command to add the module to the boot-loadable system file. This process allows the Bootstrap Loader to load the RAM-disk.

CHAPTER 4
DEVICE DRIVER PARAMETERS

Because a device driver is a collection of software routines that manages a device at a basic level, it must transform general instructions from the I/O System into device-specific instructions it then sends to the device itself. Thus, a device driver has two types of interfaces:

- An interface to the I/O System, which is the same for all device drivers.
- An interface to the device itself, which varies according to device.

This chapter discusses both of these interfaces.

4.1 I/O SYSTEM INTERFACES

The interface between the device driver and the I/O System consists of two data structures: the device-unit information block (DUIB) and the I/O request/result segment (IORS).

4.1.1 Device-Unit Information Block (DUIB)

The DUIB is an interface between a device driver and the I/O System, in that the DUIB contains the addresses of one of the following sets of routines:

- The device driver routines (in the case of custom device drivers).
- The device driver support routines (in the case of terminal drivers, common drivers, and random access drivers).

By accessing the DUIB for a unit, the I/O System can call the appropriate device driver or device driver support routine. All devices, no matter how diverse, use this standard interface to the I/O System. You must provide a DUIB for each device-unit in your hardware system. You supply the information for your DUIBs as part of the configuration process.

4.1.1.1 DUIB Structure

This section lists the elements that make up a DUIB. For devices supported by the Interactive Configuration Utility (ICU), the ICU creates DUIBs based on information you supply. For devices not supported by the ICU, you must create the DUIBs.

When creating DUIBs, code them in the format shown here (as assembly-language structures). The ICU includes your DUIB file in the assembly of ?IDEVC.A28, a Basic I/O System configuration file the ICU creates (the ? means that the first character of the file can vary). /RMX286/IOS/IDEVCF.INC contains the definition of the DUIB structure.

```
DEFINE_DUIB  <
&   NAME (14),           ;BYTE (14)
&   FILE$DRIVERS,        ;WORD
&   FUNCTS,              ;BYTE
&   FLAGS,               ;BYTE
&   DEV$GRAN,            ;WORD
&   DEV$SIZE,            ;DWORD
&   DEVICE,              ;BYTE
&   UNIT,                ;BYTE
&   DEV$UNIT,            ;WORD
&   INIT$IO,             ;WORD
&   FINISH$IO,           ;WORD
&   QUEUE$IO,            ;WORD
&   CANCEL$IO,           ;WORD
&   DEVICE$INFO$P,       ;POINTER
&   UNIT$INFO$P,         ;POINTER
&   UPDATE$TIMEOUT,      ;WORD
&   NUM$BUFFERS,         ;WORD
&   PRIORITY,            ;BYTE
&   FIXED$UPDATE,        ;BYTE
&   MAX$BUFFERS,         ;BYTE
&   RESERVED,            ;BYTE
&   >
```

where

NAME — A 14-BYTE array specifying the name of the DUIB. This name uniquely identifies the device-unit to the I/O System. Use only the first 13 bytes. The fourteenth is used by the I/O System. If the name is less than 14 characters, it is extended with spaces.

You assign the name when configuring your application system. Later, you specify the DUIB name when attaching a unit via the RQ$A$PHYSICAL$ATTACH$DEVICE system call. Device drivers do read or write this field.

FILE$DRIVERS — WORD specifying file driver validity. Setting bit number "i" of this word implies that the corresponding file driver can attach this device-unit. Clearing bit number "i" implies that the file driver cannot attach this device-unit.

The low-order bit is bit 0. The bits are associated with the file drivers as follows:

Bit "i"	File Driver
0	physical
1	stream
3	named

The remaining bits of the word must be set to zero. Device drivers do not read or write this field.

FUNCTS

BYTE specifying the I/O function validity for this device-unit. Setting bit number "i" implies that the device-unit supports the corresponding function. Clearing bit number "i" implies that the device-unit does not support the function. The low-order bit is bit 0. The bits are associated with the functions as follows:

Bit "i"	Function
0	read
1	write
2	seek
3	special
4	attach device
5	detach device
6	open
7	close

Bits 4 and 5 should always be set. Every device driver requires these functions.

This field is used for informational purposes only. Setting or clearing bits in this field does not limit the device driver from performing any I/O function. In fact, each device driver must be able to support any I/O function, either by performing the function or by returning a condition code indicating the inability of the device to perform that function. However, to provide accurate status information, this field should indicate the device's ability to perform the I/O functions. Device drivers do not read or write this field.

FLAGS

BYTE specifying characteristics of diskette devices. The significance of the bits is as follows, with bit 0 being the low-order bit:

Bit	Meaning
0	0 = bits 1-7 are undefined 1 = bits 1-7 are defined as follows
1	0 = single density 1 = double density

2	0 = single sided 1 = double sided
3	0 = 8-inch diskettes 1 = 5 1/4-inch diskettes
4	0 = standard diskette, (track 0 is single-density with 128-byte sectors) 1 = not a standard diskette or not a diskette
5-7	Reserved; must be set to zero.

If bit 4 is set to 0, then a driver for the device has the information it needs to read device information from track 0. Refer to Appendix C for more information about the format of standard diskettes.

DEV$GRAN	WORD specifying the device granularity, in bytes. This parameter applies to random access devices, and to some common devices such as tape drives. It specifies the minimum number of bytes of information the device reads or writes in one operation. If the device is a disk, magnetic bubble device, or tape drive, you should set this field equal to the sector size for the device. Otherwise, set this field equal to zero.
DEV$SIZE	DWORD specifying the number of bytes of information the device-unit can store.
DEVICE	BYTE specifying the device number of the device with which this device-unit is associated. Device drivers do not access this field.
UNIT	BYTE specifying the unit number of this device-unit. This distinguishes the unit from the other units of the device.
DEV$UNIT	WORD specifying the device-unit number. This number distinguishes the device-unit from the other units in the entire hardware system. Device drivers can ignore this field.
INIT$IO	WORD specifying the offset address of the Initialize I/O procedure associated with this unit (the base portion is the same as the base of the DEVICE$INFO$P pointer). For user-written custom device drivers, the user must supply this procedure (and the FINISH$IO, QUEUE$IO, and CANCEL$IO procedures). For common, random access, and terminal drivers (both user-written and Intel-supplied), the procedures are supplied with the I/O System. When filling out the DUIB, enter the names of the INIT$IO, FINISH$IO, QUEUE$IO, and CANCEL$IO procedures to supply this information. Device drivers do not access this field.

FINISH$IO	WORD specifying the offset address of the Finish I/O procedure associated with this unit (the base portion is the same as the base of the DEVICE$INFO$P pointer). Device drivers do not access this field.
QUEUE$IO	WORD specifying the offset address of the Queue I/O procedure associated with this unit (the base portion is the same as the base of the DEVICE$INFO$P pointer). Device drivers do not access this field.
CANCEL$IO	WORD specifying the offset address of the Cancel I/O procedure associated with this unit (the base portion is the same as the base of the DEVICE$INFO$P pointer). Device drivers do not access this field.
DEVICE$-INFO$P	POINTER to a structure containing additional information about the device. The common, random access, and terminal device drivers require, for each device, a Device Information Table, in a particular format. Chapter 5 lists the structure of the Device Information Table used with common and random access drivers. Chapter 6 lists the structure used with terminal drivers. If you are writing a custom driver, you can place information in the Device Information Table according to the needs of your driver. Specify a zero for this parameter if the associated device driver does not use this field.
UNIT$-INFO$P	POINTER to a structure containing additional information about the unit. Random access and terminal device drivers require this Unit Information Table in a particular format. Chapter 5 describes the format for random access drivers. Chapter 6 describes the format for terminal drivers. If you are writing a custom device driver, place information in this structure, depending on the needs of your driver. Specify a zero for this parameter if the associated device driver does not use this field.
UPDATE$-TIMEOUT	WORD specifying the number of system time units the I/O System must wait before writing a partial sector after processing a write request for a disk device. In the case of drivers for devices that are neither disk nor magnetic bubble devices, set this field to 0FFFFH during configuration. This field applies only to the device-unit specified by this DUIB, and is independent of updating done either because of the value in the FIXED$UPDATE field of the DUIB or by means of the I/O System A$UPDATE system call. Device drivers do not access this field.

NUM$-BUFFERS	WORD that, if not zero, both specifies the device is a random access device and indicates the number of buffers of device-granularity size the I/O System allocates. The I/O System uses these buffers to perform data blocking and deblocking operations. That is, it guarantees that data is read or written beginning on sector boundaries. If you desire, the random access support routines can also guarantee that no data is written or read across track boundaries in a single request (see the section on the Unit Information Table in Chapter 3). A value of zero indicates the device is not a random access device. Device drivers do not access this field.
PRIORITY	BYTE specifying the priority of the I/O System service task for the device. Device drivers do not access this field.
FIXED$-UPDATE	BYTE indicating whether the fixed update option was selected for the device-unit when the application system was configured. This option, when selected, causes the I/O System to finish any write requests that had not been finished earlier because less than a full sector remained to be written. Fixed updates are performed throughout the entire system whenever a time interval (specified during configuration) elapses. This is independent of the updating indicated for a particular device (by the UPDATE$TIMEOUT field of the DUIB) or the updating of a particular device indicated by the A$UPDATE system call of the I/O System. A value of 0FFH indicates fixed updating has been selected for this device; a value of zero indicates it has not been selected. Device drivers do not access this field.
MAX$-BUFFERS	BYTE specifying the maximum number of buffers the Extended I/O System can allocate for a connection to this device-unit when the connection is opened by a call to S$OPEN. The value in this field is specified during configuration. Device drivers do not access this field.
RESERVED	Intel reserves this BYTE for future use.

4.1.1.2 Using the DUIBs

To use the I/O System to communicate with files on a device-unit, you must first attach the unit by invoking the RQ$A$PHYSICAL$ATTACH$DEVICE system call (refer to the *Extended iRMX II Basic I/O System Calls* manual for a description of this system call).

When you attach a unit, the I/O System assumes the device-unit identified by the device name field of the DUIB has the characteristics identified in the remainder of the DUIB. Thus, whenever the application software makes an I/O request via the connection to the attached device-unit, the I/O System ascertains the characteristics of that unit by examining the associated DUIB. The I/O System looks at the DUIB and calls the appropriate device driver or device driver support routines listed there to process the I/O request.

If you want the I/O System to assume different characteristics at different times for a particular device-unit, you can supply multiple DUIBs, each containing identical device number, unit number, and device-unit number parameters, but different DUIB name parameters. Then you can select one of these DUIBs by specifying the appropriate dev$name parameter in the RQ$A$PHYSICAL$ATTACH$DEVICE system call. However, before you can switch the DUIBs for a unit, you must detach the unit.

Figure 4-1 illustrates this concept. It shows six DUIBs, two for each of three units of one device. The main difference within each pair of DUIBs in this figure is the device granularity parameter, which is either 128 or 512. With this setup, a user can attach any unit of this device with one of two device granularities. In Figure 4-1, units 0 and 1 are attached with a granularity of 128 and unit 2 with a granularity of 512. To change this, the user can detach the device and attach it again using the other DUIB name.

NOTE

When the I/O System accesses a device containing named files, it obtains information such as granularity, density, size (5-1/4" or 8" for diskettes), or the number of sides (single or double) from the volume label. Therefore it is not necessary to supply a different DUIB for every kind of volume you intend to use. However, for iRMX II applications, you must supply a separate DUIB for every kind of volume you intend to format via the FORMAT Human Interface command.

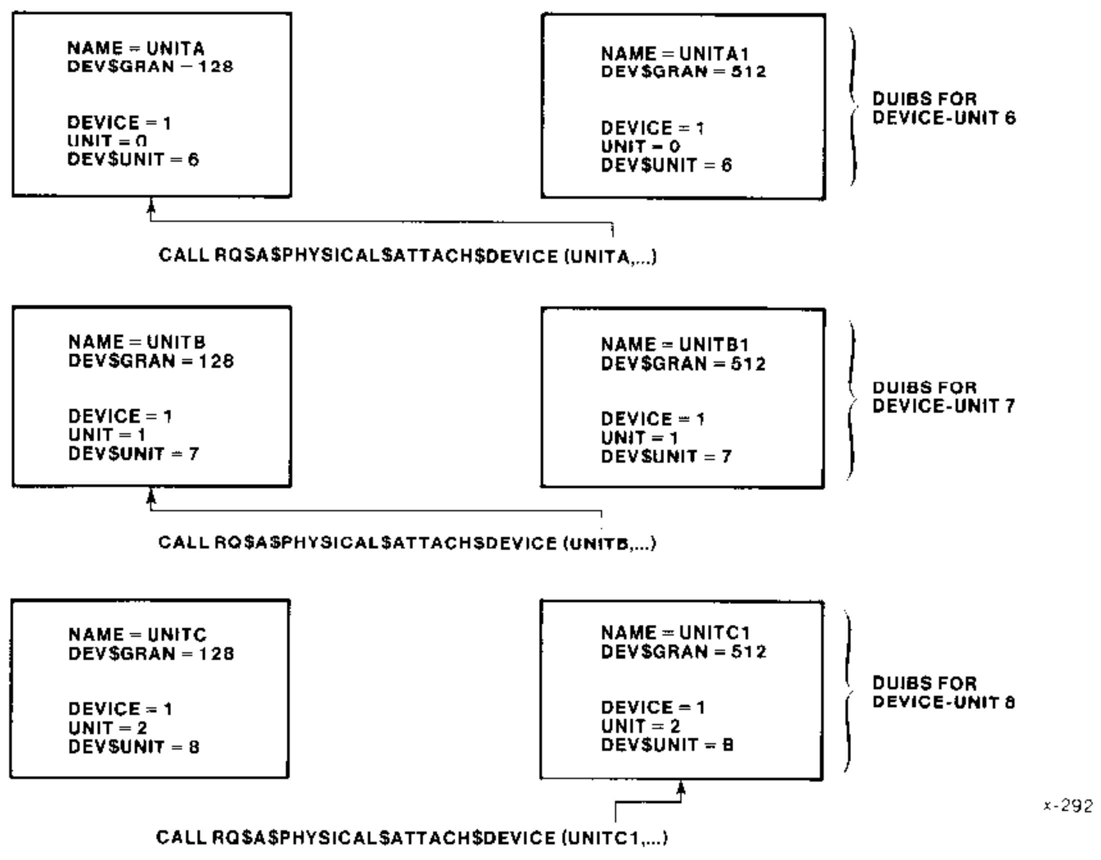


Figure 4-1. Attaching Devices

4.1.1.3 Creating DUIBs

During interactive configuration, you must provide the information for all of the DUIBs. The configuration file, which the ICU produces, sets up the DUIBs when it executes. Observe the following guidelines when supplying DUIB information:

- Specify a unique name for every DUIB, even those that describe the same device-unit.
- For every device-unit in the hardware configuration, provide information for at least one DUIB. Because the DUIB contains the addresses of the device driver routines, this guarantees that no device-unit is left without a device driver to handle its I/O.
- Make sure to specify the same device driver procedures in all of the DUIBs associated with a particular device. There is only one set of device driver routines for a given device, and each DUIB for that device must specify this unique set of routines.
- If you write a common or random access device driver, you must supply a Device Information Table for each device. If you write a random access device driver, you must also supply a Unit Information Table for each unit. See Chapter 5 for

specifications of these tables. If you are using custom device drivers and they require these or similar tables, you must supply them, as well.

- If you write a terminal driver, you must supply a terminal device information table for each terminal device driver, as well as a unit information table for each terminal. See Chapter 6 for specifications of these tables.

4.1.2 I/O Request/Result Segment (IORS)

An IORS is the second structure that forms an interface between a device driver and the I/O System. The I/O System creates an IORS when an application task requests an I/O operation. The IORS contains information about the request and about the unit on which the operation is to be performed. The I/O System passes the IORS to one of the high-level driver procedures (the Queue I/O Procedure), which then processes the request or puts it in a queue for processing. After performing the requested operation, the device driver must modify the IORS to indicate what it has done and send the IORS back to the response mailbox (exchange) indicated in the IORS.

If you are writing a custom driver, the high-level driver procedures you write (Initialize I/O, Finish I/O, Queue I/O, and Cancel I/O) must be aware of the structure of the IORS. If you are writing a common or random access driver, the procedures you write must also be aware of the IORS structure, because the high-level driver procedures (supplied by the I/O System) pass the IORS on for further processing.

If you are writing a terminal driver, your procedures do not need to be aware of the IORS. The high-level terminal driver procedures (called the Terminal Support Code) transform the information they receive from the IORS into different structures they pass on to your lower-level driver procedures. Refer to Chapter 6 for information about the structures used by terminal drivers.

The IORS is structured as follows:

```
    DECLARE
        IORS            STRUCTURE(
            STATUS          WORD,
            UNIT$STATUS     WORD,
            ACTUAL          WORD,
            ACTUAL$FILL     WORD,
            DEVICE          WORD,
            UNIT            BYTE,
            FUNCT           BYTE,
            SUBFUNCT        WORD,
            DEV$LOC         DWORD,
            BUFF$P          POINTER,
            COUNT           WORD,
            COUNT$FILL      WORD,
            AUX$P           POINTER,
            LINK$FOR        POINTER,
            LINK$BACK       POINTER,
            RESP$MBOX       TOKEN,
            DONE            BYTE,
            FILL            BYTE,
            CANCEL$ID       TOKEN,
            CONN$T          TOKEN);
```

where

STATUS — WORD in which the device driver must place the condition code for the I/O operation. The E$OK condition code indicates successful completion of the operation. For a complete list of possible condition codes, see either the *Extended iRMX II NUCLEUS SYSTEM CALLS* manual, the *Extended iRMX II BASIC I/O System Calls* manual, or the *Extended iRMX II Extended I/O System Calls* manual.

UNIT$-STATUS — WORD in which the device driver must place additional status information if the status parameter was set to indicate the E$IO condition. The unit status codes and their descriptions are as follows:

Code	Mnemonic	Description
0	IO$UNCLASS	Unclassified error
1	IO$SOFT	Soft error; a retry is possible
2	IO$HARD	Hard error; a retry is impossible
3	IO$OPRINT	Operator intervention is required (the device is offline)
4	IO$WRPROT	Write-protected volume

Code	Mnemonic	Description
5	IO$NO$DATA	No data on the next tape record
6	IO$MODE	A read (or write) was attempted before the previous write (or read) completed.
7	IO$NOSPARES	The number of bad tracks or sectors exceeds the number of alternates available.
8	IO$ALT$AS-SIGNED	An alternate track or sector was assigned to replace a defective one.

The I/O System reserves bits 0 through 3 (the least significant four bits) of this field for unit status codes. The high 12 bits of this field can be used for any other purpose. For more information about the data returned by your device controller, refer to the hardware reference manual for your controller.

ACTUAL	WORD the device driver must update upon completion of an I/O operation to indicate the number of bytes of data actually transferred.
ACTUAL$FILL	Reserved WORD.
DEVICE	WORD into which the I/O System places the number of the device for which this request is intended.
UNIT	BYTE into which the I/O System places the number of the unit for which this request is intended.
FUNCT	BYTE into which the I/O System places the function code for the operation to be performed. Possible function codes are

Code	Function
0	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE

SUBFUNCT	WORD into which the I/O System places the actual function code of the operation, when the F$SPECIAL function code was placed into the FUNCT field. The value in this field depends on the file driver being used with this device. The possible subfunctions and the driver types to which they apply are as follows:

File Driver For Connection	Subfunct Value	Function
Physical	0	Format track
Stream	0	Query
Stream	1	Satisfy
Physical or Named	2	Notify
Physical	3	Get disk/tape data
Physical	4	Get terminal data
Physical	5	Set terminal data
Physical	6	Set signal
Physical	7	Reset (rewind tape or reset disk)
Physical	8	Read tape file mark
Physical	9	Write tape file mark
Physical	10	Retension tape
	11	Reserved
Physical	12	Set bad track information
Physical	13	Get bad track information
	14-32767	Reserved for other Intel products

The values from 32768 to 65535 are available for user-written/custom device drivers when used with the physical file driver.

DEV$LOC

DWORD into which the I/O System initially places the absolute byte location on the I/O device where the operation is to be performed. For example, for a write operation, this is the address on the device where writing begins. The I/O System fills out this information when it passes the IORS to the driver or the driver support routines.

If the device driver is a random access driver, the random access support routines modify the information in the DEV$LOC field before passing the IORS on to driver procedures listed in Chapter 5. The value the random access support routines fill out depends upon the TRACK$SIZE field in the unit's Unit Information Table (see Chapter 5).

- If the TRACK$SIZE field is zero, the random access support routines divide the value in DEV$LOC by the device granularity and place that value (the absolute sector number) in the DEV$LOC field.

- If the TRACK$SIZE field is nonzero, the random access support routines divide the absolute byte number in DEV$LOC by TRACK$SIZE to calculate the track and sector numbers. The routines then place the track number in the high-order WORD (of DEV$LOC) and the sector number in the low-order WORD (of DEV$LOC).

BUFF$P	POINTER the I/O System sets to indicate the internal buffer where data is read from or written to.
COUNT	WORD the I/O System sets to indicate the number of bytes to transfer.
COUNT$FILL	Reserved WORD.
AUX$P	POINTER the I/O System can set to indicate the location of auxiliary data. Normally, the I/O System uses AUX$P to pass or receive the additional data the various subfunctions of the SPECIAL call require.

The following paragraphs define the particular data structures pointed to by AUX$P. The data structure actually pointed to depends upon the SUBFUNCT field of the IORS.

In a request to format a track on a disk or diskette, FUNCT equals f$special, SUBFUNCT equals format track (0), and AUX$P points to a structure of the form:

```
DECLARE FORMAT$TRACK STRUCTURE(
                         TRACK$NUMBER    WORD,
                         INTERLEAVE      WORD,
                         TRACK$OFFSET    WORD,
                         FILL$CHAR       BYTE);
```

These fields are defined as follows:

track$number is the number of the track to be formatted. Acceptable values are 0 to (number of tracks on the volume - 1).

interleave is the interleave factor for the track. (That is, the number of physical sectors to advance when locating the next logical sector.) The supplied value, before being used, is evaluated MOD the number of sectors per track.

track$offset is the number of physical sectors to advance when locating the first logical sector on the next track.

fill$char is the BYTE value with which each sector is to be filled.

In a request to set up a mailbox in which the device driver sends an object whenever a door to a flexible disk drive is opened or the STOP button on a hard disk drive is pressed, FUNCT equals f$special, SUBFUNCT equals notify (2), and AUX$P points to a structure of the form:

```
DECLARE SETUP$NOTIFY STRUCTURE(
                    MAILBOX      TOKEN,
                    OBJECT       TOKEN);
```

where the fields are defined in the *Extended iRMX II Basic I/O System Calls* manual in the description of the A$SPECIAL system call. Random access drivers do not need to create the mailbox, but they must send the object to the mailbox whenever they notice a media change.

In a request to obtain information about iSBC 215G (supported) disk devices, FUNCT equals f$special, SUBFUNCT equals get device characteristics (3), and AUX$P points to a structure of the form:

```
DECLARE DISK$DRIVE$DATA STRUCTURE(
                    CYLINDERS       WORD,
                    FIXED           BYTE,
                    REMOVABLE       BYTE,
                    SECTORS         BYTE,
                    SECTOR$SIZE     WORD,
                    ALTERNATES      BYTE);
```

where the fields are defined in the *Extended iRMX II Basic I/O System Calls* manual in the description of the A$SPECIAL system call.

In a request to obtain information about iSBX 217C tape drives (associated with an iSBC 215G board) or iSBC 214 tape drives, FUNCT equals f$special, SUBFUNCT equals get device characteristics (3), and AUX$P points to a structure of the form:

```
DECLARE TAPE$DRIVE$DATA STRUCTURE(
                    TAPE            BYTE,
                    RESERVED(7)     BYTE);
```

where the fields are defined in the *Extended iRMX II Basic I/O System Calls* manual in the description of the A$SPECIAL system call.

In a request to read or write terminal mode information for a terminal being driven by a terminal driver, FUNCT equals f$special, SUBFUNCT equals get terminal attributes (4) for reading or set terminal attributes (5) for writing, and AUX$P points to a structure of the form:

```
DECLARE TERMINAL$ATTRIBUTES STRUCTURE(
                    NUM$WORDS           WORD,
                    NUM$USED            WORD,
                    CONNECTION$FLAGS    WORD,
                    TERMINAL$FLAGS      WORD,
                    IN$BAUD$RATE        WORD,
                    OUT$BAUD$RATE       WORD,
                    SCROLL$LINES        WORD,
                    X$Y$SIZE            WORD,
                    X$Y$OFFSET          WORD,
                    SPECIAL$MODES       WORD,
                    HIGH$WATER$MARK     WORD,
                    LOW$WATER$MARK      WORD,
                    FC$ON$CHAR          WORD,
                    FC$OFF$CHAR         WORD,
                    LINK$PARAMETER      WORD,
                    SPC$HI$WATER$MARK   WORD,
                    SPECIAL$CHAR(4)     BYTE);
```

where the fields are defined in the *Extended iRMX II Basic I/O System Calls* manual in the description of the A$SPECIAL system call. If you are using the Terminal Support Code, your driver does not need to contain special code to support this subfunction.

In a request to set up special character recognition in the input stream of a terminal driver for signaling purposes, FUNCT equals f$special, SUBFUNCT equals signal (6), and AUX$P points to a structure of the form:

```
DECLARE SIGNAL$CHARACTER STRUCTURE(
                    SEMAPHORE       TOKEN,
                    CHARACTER       BYTE);
```

where the fields are defined in the *Extended iRMX II Basic I/O System Calls* manual in the description of the A$SPECIAL system call. If you are using the Terminal Support Code, your driver does not need to contain special code to support this subfunction.

In a request to read a tape file mark, FUNCT equals f$special, SUBFUNCT equals read tape file mark (8), and AUX$P points to a structure of the form:

```
DECLARE READ$FILE$MARK STRUCTURE(
                          SEARCH              BYTE);
```

where the field is defined in the *Extended iRMX II Basic I/O System Calls* manual in the description of the A$SPECIAL system call.

In a request to set or get bad track/sector information, FUNCT equals f$special, SUBFUNCT equals set bad track/sector information (12) or get bad track/sector information (13), and AUX$P points to a structure of the following form:

```
DECLARE BAD$TRACK$INFO  STRUCTURE(
                          RESERVED            WORD,
                          COUNT               WORD,
                          BAD$TRACKS(255)     DWORD),

BAD$TRACKS(255)  STRUCTURE(
                          CYLINDER            WORD,
                          HEAD                BYTE,
                          SECTOR              BYTE)
AT (@BAD$TRACK$INFO.BAD$TRACKS);
```

where the fields are defined in the *Extended iRMX II Basic I/O System Calls* manual in the description of the A$SPECIAL system call.

LINK$FOR	POINTER the device driver or device driver support routines can use to implement a request queue. This field points to the location of the next IORS in the queue.
LINK$BACK	POINTER the device driver or device driver support routines can use to implement a request queue. This field points to the location of the previous IORS in the queue.
RESP$MBOX	TOKEN the I/O System fills with a token for the response mailbox. On completion of the I/O request, the device driver or device driver support routines must send the IORS to this response mailbox or exchange.
DONE	BYTE the device driver can set to TRUE (0FFH) or FALSE (00H) to indicate whether the entire request has been completed.
FILL	Reserved BYTE.
CANCEL$ID	TOKEN the I/O System fills in to identify queued I/O requests the Cancel I/O procedure can remove from the queue. For I/O operations that require multiple requests (and therefore multiple IORSs), the I/O System uses the same CANCEL$ID value in all IORSs for that operation. This allows the Cancel I/O procedure to remove all IORSs for a given operation.

CONN$T	TOKEN used in requests to the I/O System. This field contains the token of the file connection through which the request was issued.

4.2 DUIB AND IORS FIELDS USED BY DEVICE DRIVERS

Tables 4-1, 4-2, and 4-3 indicate, for common, random access, and custom drivers, the fields of DUIBs and IORSs with which user-written portions of device drivers need to be concerned. The user-written portions of terminal drivers do not need to use either of these structures. Device drivers should not write information to these fields unless specifically indicated in the tables.

Table 4-1. DUIB and IORS Fields Used by Common Device Drivers

	Attach Device	Detach Device	Open	Close	Read	Write	Seek	Special
DUIB								
Name								
File$drivers								
Functs								
Flags	m	m	m	m	m	m	m	m
Dev$gran	m	m	m	m	m	m	m	m
Dev$size	m	m	m	m	m	m	m	m
Device								
Unit	m	m	m	m	m	m	m	m
Dev$unit								
Init$io								
Finish$io								
Queue$io								
Cancel$io								
Device$info$p	m	m	m	m	m	m	m	m
Unit$info$p	m	m	m	m	m	m	m	m
Update$timeout								
Num$buffers								
Priority								
Fixed$update								
Max$buffers								
IORS								
Status	w	w	w	w	w	w	w	w
Unit$status	w	w	w	w	w	w	w	w
Actual					w	w		
Actual$fill								
Device								
Unit	m	m	m	m	m	m	m	m
Funct	r	r	r	r	r	r	r	r
Subfunct								r
Dev$loc					m	m	m	
Buff$p					r	r		
Count					r	r		
Count$fill								
Aux$p								m
Link$for								
Link$back								
Resp$mbox								
Done	w	w	w	w	w	w	w	w
Fill								
Cancel$id								
Conn$t								

r--is read by the device driver

w--is written by the device driver

m--might be read by some device drivers

Table 4-2. DUIB and IORS Fields Used by Random Access Device Drivers

	Attach Device	Detach Device	Open	Close	Read	Write	Seek	Special
DUIB								
Name								
File$drivers								
Functs								
Flags	m	m	m	m	m	m	m	m
Dev$gran	m	m	m	m	m	m	m	m
Dev$size	m	m	m	m	m	m	m	m
Device								
Unit	m	m	m	m	m	m	m	m
Dev$unit								
Init$io								
Finish$io								
Queue$io								
Cancel$io								
Device$info$p	m	m	m	m	m	m	m	m
Unit$info$p	m	m	m	m	m	m	m	m
Update$timeout								
Num$buffers								
Priority								
Fixed$update								
Max$buffers								
IORS								
Status	w	w	w	w	w	w	w	w
Unit$status	w	w	w	w	w	w	w	w
Actual					w	w		
Actual$fill								
Device								
Unit	m	m	m	m	m	m	m	m
Funct	r	r	r	r	r	r	r	r
Subfunct								r
Dev$loc						r	r	r
Buff$p						r	r	
Count						r	r	
Count$fill								
Aux$p								m
Link$for								
Link$back								
Resp$mbox								
Done	w	w	w	w	w	w	w	w
Fill								
Cancel$id								
Conn$t								

r--is read by the device driver

w--is written by the device driver

m--might be read by some device drivers

Table 4-3. DUIB and IORS Fields Used by Custom Device Drivers

	Attach Device	Detach Device	Open	Close	Read	Write	Seek	Special
DUIB								
Name								
File$drivers								
Functs								
Flags	m	m	m	m	m	m	m	m
Dev$gran	m	m	m	m	m	m	m	m
Dev$size	m	m	m	m	m	m	m	m
Device								
Unit	m	m	m	m	m	m	m	m
Dev$unit								
Init$io								
Finish$io								
Queue$io								
Cancel$io								
Device$info$p	m	m	m	m	m	m	m	m
Unit$info$p	m	m	m	m	m	m	m	m
Update$timeout								
Num$buffers								
Priority								
Fixed$update								
Max$buffers								
IORS								
Status	w	w	w	w	w	w	w	w
Unit$status	w	w	w	w	w	w	w	w
Actual								
Actual$fill								
Device								
Unit	m	m	m	m	m	m	m	m
Funct	r	r	r	r	r	r	r	r
Subfunct								
Dev$loc					m	m	m	
Buff$p					r	r		
Count					r	r		
Count$fill								
Aux$p								
Link$for	a	a	a	a	a	a	a	a
Link$back	a	a	a	a	a	a	a	a
Resp$mbox	r	r	r	r	r	r	r	r
Done	a	a	a	a	a	a	a	a
Fill	a	a	a	a	a	a	a	a
Cancel$id				m				
Conn$t								

r--is read by the device driver

w--is written by the device driver

m--might be read by some device drivers

a--available for any purpose suiting the needs of the device driver

4.3 DEVICE INTERFACES

To carry out I/O requests, one or more of the routines in every device driver must actually send commands to the device itself. The steps a procedure of this sort must go through vary considerably, depending on the type of I/O device. Some devices are controlled by on-board firmware the driver communicates with by sending firmware commands and receiving status. Other devices may require different methods. The I/O System places no restrictions on the method of communicating with devices. Use the method the device requires.

However, a consideration that applies to many device drivers is the manner in which devices expect addresses to be presented. For example, the Operating System expects all addresses to be logical addresses of this form:

```
selector:offset
```

On the other hand, most device controllers expect addresses to be physical (24-bit) addresses. To compensate for this difference, a device driver must convert from one form of address to another when sending or receiving device information.

For example, writing information to a device usually involves giving the controller the address of the data buffer that holds the information. To the device driver (or any other program that fills the buffer), the buffer is known by its logical address. But the controller expects the 24-bit physical address of the buffer. Therefore, the driver must convert the buffer's logical address to a physical address before passing the address to the device controller.

The Operating System provides two ways of converting a logical address into a physical address. The Nucleus provides one method with the system call RQE$GET$ADDRESS. The Basic I/O System provides a similar but faster method for use by device drivers.

The Basic I/O System method involves a procedure called BIOS$GET$ADDRESS. This procedure is located in the /RMX286/IOS/XDRVUT.LIB library. When you link your driver code to this library and call the BIOS$GET$ADDRESS procedure, the procedure converts logical addresses to physical addresses. Because this conversion program is a procedure, not a system call, it runs in the calling program's environment without invoking other Basic I/O System routines.

The calling sequence for this procedure is as follows:

```
physical = BIOS$GET$ADDRESS (logical, except$ptr);
```

where

physical	A DWORD returned by the procedure that contains the 24-bit physical address desired. The high-order byte of this DWORD is set to zero.

logical	A POINTER specifying the logical address to be converted. The pointer must be in the form selector:offset.
except$ptr	A POINTER to a WORD containing a condition code returned by the procedure. Possible condition codes include

E$OK	No exceptional conditions occurred.
E$BAD$-ADDR	The logical address is invalid. Either the selector does not point to a valid segment, or the offset is outside the segment boundaries.

The following example illustrates how a program declares and invokes BIOS$GET$ADDRESS:

```
$INCLUDE(/rmx286/inc/rmxplm.ext)    /* Declares all system calls */

    DECLARE phys$addr   DWORD;
    DECLARE buff$ptr    POINTER;
    DECLARE status$ptr  POINTER;

BIOS$GET$ADDRESS: PROCEDURE(log$addr, except$ptr) DWORD EXTERNAL;

    DECLARE (log$addr, except$ptr) POINTER

END BIOS$GET$ADDRESS;

SAMPLE_PROCEDURE:
    PROCEDURE;

        •
        •      Typical PL/M-286 Statements
        •

phys$addr = BIOS$GET$ADDRESS(buff$ptr, status$ptr);

        •
        •   Typical PL/M-286 Statements
        •

END SAMPLE_PROCEDURE;
```

Converting from physical addresses to logical addresses is also necessary if you need to have access to the information returned by a device controller. The Nucleus provides a system call named RQE$CREATE$DESCRIPTOR that sets up an entry in the descriptor table for any segment whose physical address and size you specify. By setting up a descriptor, you allow programs to access that memory with logical addresses.

CHAPTER 5
WRITING COMMON OR RANDOM ACCESS DEVICE DRIVERS

This chapter describes how to write device drivers for common and random access devices. It lists the procedures the I/O System supplies, describes the data structures that must exist, and includes the calling sequences for the procedures you must provide when writing a common or random access device driver. Where possible, descriptions of the duties of these procedures accompany the calling sequences.

In addition, this chapter describes the purpose and calling sequence for five I/O System-supplied procedures that random access drivers call under certain conditions.

Throughout this chapter, differences between message-passing and interrupt-driven devices are noted by the following conventions (refer to Appendixes A and B for descriptions of these devices. Message-passing data structures and parameter descriptions are shaded in gray where they differ from interrupt-driven data structures and parameter descriptions. The terms **interrupt** and **message** (and variations of these terms) are noted as "interrupt/message" where they mean that interrupt-driven devices use an interrupt while message-passing devices use a message to accomplish the same purpose.

5.1 I/O SYSTEM-SUPPLIED ROUTINES

For common and random access devices, the I/O System supplies the highest-level procedures. It calls these procedures when processing I/O requests. Flow charts and complete descriptions for these procedures appear in Appendixes A (interrupt-driven devices) and B (message-passing devices). The names of these procedures and the general operations they perform are as follows:

Procedure	Function
INIT$IO	Creates the resources needed by the remainder of the driver routines, creates an interrupt/message task, and calls a user-supplied routine that initializes the device itself.
FINISH$IO	Deletes the resources used by the other driver routines, deletes the interrupt/message task, and calls a user-supplied procedure that performs final processing on the device itself.

QUEUE$IO	Places I/O requests (IORSs) on a queue of requests. This procedure starts the device processing the first request on the queue.
CANCEL$IO	Removes one or more requests from the request queue, possibly stopping the processing of a request that has already been started.

For Intel-supplied common or random access drivers, these high-level procedures call other Intel-supplied procedures that communicate directly with specific devices. If you write your own common or random access device driver, these high-level procedures call procedures that you must write. A later section of this chapter describes the interfaces to these user-written procedures.

The INIT$IO, FINISH$IO, QUEUE$IO, and CANCEL$IO procedures process I/O requests for both common and random access devices. They distinguish between types of devices based on the value of the NUM$BUFFERS field in the unit's device-unit information block (DUIB). (When calling each of these routines, the I/O System supplies a pointer to the DUIB as one of the parameters.) If the NUM$BUFFERS field is nonzero, the routines assume the device is a random access device. If the NUM$BUFFERS field is zero, the routines assume the device is a common device.

5.1.1 Interrupt Task

In addition to the above routines, the I/O System supplies an interrupt handler and an interrupt task (called INTERRUPT$TASK) for interrupt-driven devices. The interrupt handler and interrupt task respond to all interrupts generated by the units of a device, process those interrupts, and start the device working on the next I/O request on the queue. The INIT$IO procedure creates the interrupt task.

After a device finishes processing a request, it sends an interrupt to the processor. As a consequence, the processor calls the interrupt handler. This handler invokes the SIGNAL$INTERRUPT system call to tell a waiting interrupt task to process the interrupt. The handler doesn't process the interrupt itself because it is limited in the types of system calls it can make and the number of interrupts that can be enabled while it is processing.

The interrupt task feeds the results of the interrupt back to the I/O System (data from a read operation, status from other types of operations). The interrupt task then gets the next I/O request from the queue and starts the device processing this request. This cycle continues until the device is detached.

Figure 5-1 shows the interaction between an interrupt task, an I/O device, an I/O request queue, and the QUEUE$IO device driver procedure. The interrupt task in this figure is in a continual cycle of waiting for an interrupt, processing it, getting the next I/O request, and starting up the device again. While this is going on, the QUEUE$IO procedure runs in parallel, putting additional I/O requests on the queue.

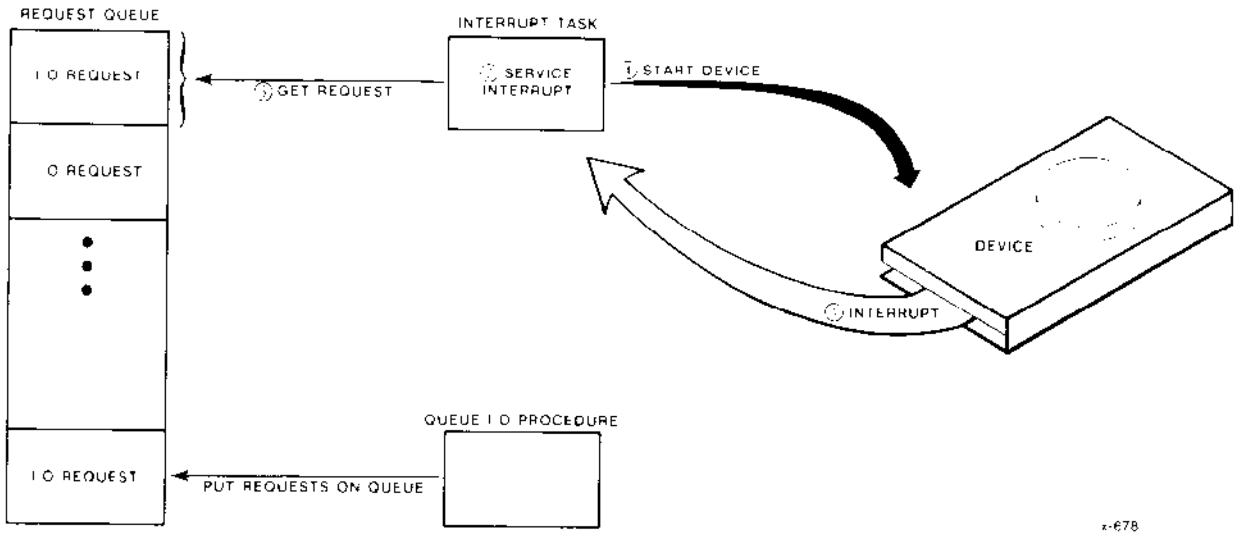
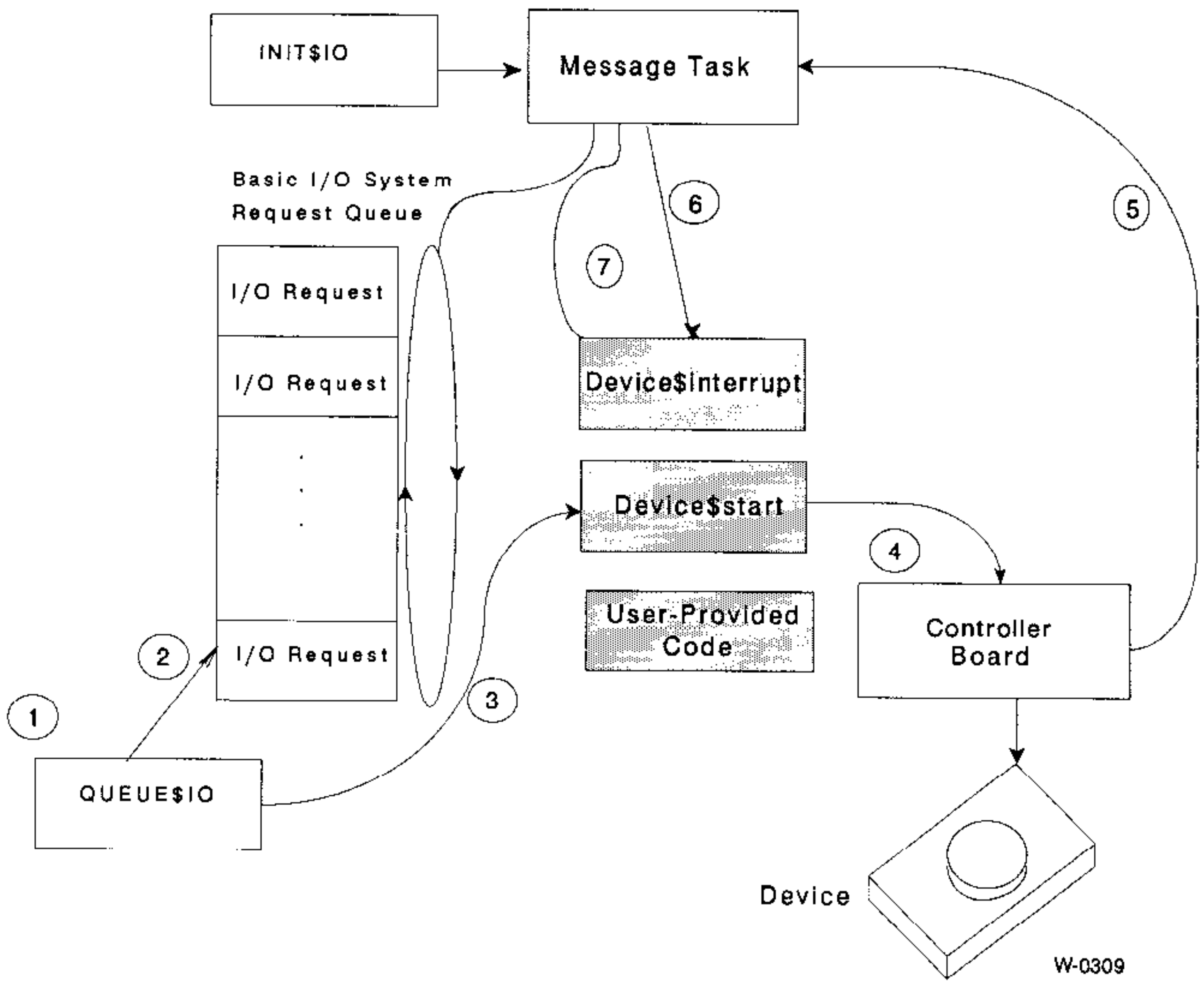


Figure 5-1. Interrupt Task Interaction

5.1.2 Message Task

In addition to INIT$IO, QUEUE$IO, CANCEL$IO, and FINISH$IO, the I/O System supplies a message task (called MESSAGE$TASK) for message-passing devices. The message task responds to all messages generated by the units of a device, processes those messages, and starts the device working on the unstarted I/O requests on the queue.

Figure 5-2 shows the interaction between a message task, an I/O device, an I/O request queue, the QUEUE$IO device driver procedure, and user-supplied driver procedures. The message task running on the CPU board is in a continual cycle of waiting for a message, processing it, then checking the next request on the I/O request queue. If the request is unstarted, the message task starts the device processing the request. If the request is marked DONE, the message task removes it from the queue. While the message task goes through this cycle, the QUEUE$IO procedure runs in parallel, putting additional I/O requests on the queue.



1. An I/O request comes in to the QUEUE$IO procedure.
2. The QUEUE$IO procedure places the request on the I/O request queue.
3. The QUEUE$IO procedure calls the user-supplied device start procedure.
4. The device start procedure sends a message to the controller board.
5. After processing this device driver request, the controller board sends a message to the message task.
6. The message task calls the user-supplied device interrupt procedure that tracks which IORS corresponds to each transaction ID. It also marks the I/O request as DONE, when the I/O request is complete. If the I/O request is complete, the message task returns the IORS to the user who originated the request.
7. The message task calls the device start procedure to start the next available unstarted request on the I/O request queue. The message task waits for a message from the controller.

Figure 5-2. Message Task Interaction

5.2 I/O SYSTEM ALGORITHM FOR CALLING DRIVER PROCEDURES

The I/O System calls each of the four device driver procedures (INIT$IO, QUEUE$IO, CANCEL$IO, and FINISH$IO) in response to specific conditions. Figure 5-3 is a flow chart that illustrates the conditions under which three of the four procedures are called. The following numbered paragraphs discuss the portions of Figure 5-3 labeled with corresponding circled numbers.

1. To start I/O processing, an application task must make an I/O request. It can do this by invoking any of a variety of system calls. However, the first I/O request to each device-unit must be an RQ$A$PHYSICAL$ATTACH$DEVICE system call.
2. If the request results from an RQ$A$PHYSICAL$ATTACH$DEVICE system call, the I/O System checks to see if any other units of the device are currently attached. If no other units of the device are currently attached, the I/O System realizes that the device has not been initialized and calls the INIT$IO procedure first, before queuing the request.
3. Whether or not the I/O System called the INIT$IO procedure, it calls the QUEUE$IO procedure to queue the request for execution.
4. If the request just queued resulted from an RQ$A$PHYSICAL$DETACH$DEVICE system call, the I/O System checks to see if any other units of the device are currently attached. If no other units of the device are attached, the I/O System calls the FINISH$IO procedure to do any final processing on the device and clean up resources used by the device driver routines.

The I/O System calls the fourth device driver procedure, the CANCEL$IO procedure, under the following conditions:

- If the user makes an RQ$A$PHYSICAL$DETACH$DEVICE system call specifying the hard detach option, to forcibly detach the connection objects associated with a device-unit. The *Extended iRMX II Basic I/O System Calls* manual describes the hard detach option.
- If the job containing the task which made a request is deleted.

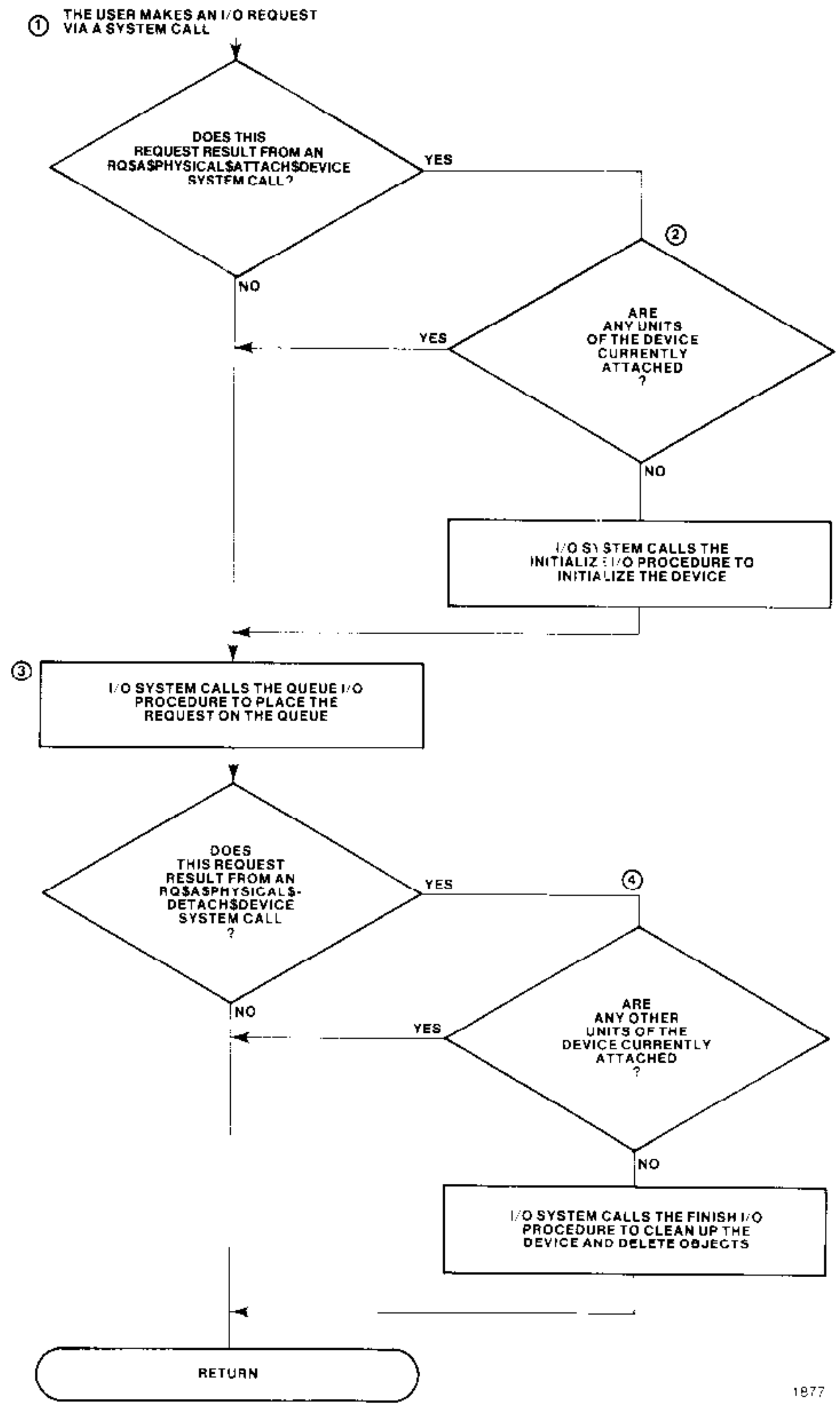

① THE USER MAKES AN I/O REQUEST VIA A SYSTEM CALL
DOES THIS REQUEST RESULT FROM AN RQ$A$PHYSICAL$ATTACH$DEVICE SYSTEM CALL?
YES
NO
②
ARE ANY UNITS OF THE DEVICE CURRENTLY ATTACHED ?
YES
NO
I/O SYSTEM CALLS THE INITIALIZE I/O PROCEDURE TO INITIALIZE THE DEVICE
③
I/O SYSTEM CALLS THE QUEUE I/O PROCEDURE TO PLACE THE REQUEST ON THE QUEUE
DOES THIS REQUEST RESULT FROM AN RQAPHYSICAL$-DETACH$DEVICE SYSTEM CALL ?
YES
NO
④
ARE ANY OTHER UNITS OF THE DEVICE CURRENTLY ATTACHED ?
YES
NO
I/O SYSTEM CALLS THE FINISH I/O PROCEDURE TO CLEAN UP THE DEVICE AND DELETE OBJECTS
RETURN
1877

Figure 5-3. How the I/O System Calls the Device Driver Procedures

5.3 REQUIRED DATA STRUCTURES

The principal data structures supporting common and random access I/O are the Device-Unit Information Block (DUIB), the Device Information Table, and the Unit Information Table (random access drivers only).

The DUIB is the primary interface between the device driver and the I/O System. Each device-unit that communicates via the I/O System has its own DUIB. Each DUIB contains one pointer to a Device Information Table and another to a Unit Information Table. Chapter 4 describes all the fields that make up the DUIB.

If you write your own common or random access driver procedures, the I/O System-supplied routines INIT$IO, FINISH$IO, QUEUE$IO, and CANCEL$IO must be able to call these routines. For this to happen, you must supply the addresses of these user-supplied routines, as well as other information, in a Device Information Table. Intel-supplied device drivers also use Device Information Tables to supply information about their lower-level routines. The "Device Information Table" section of this chapter describes the fields that make up the Device Information Table.

In addition, random access drivers require Unit Information Tables for use in processing I/O requests for devices with multiple units (such as a disk controller with multiple drives) where the units have different characteristics. The "Unit Information Table" section of this chapter describes the fields that make up the Unit Information Table.

In setting up DUIBs, those DUIBs that correspond to units of the same device should point to the same Device Information Table. But they should point to different Unit Information Tables if the units have different characteristics. Figure 5-4 illustrates this.

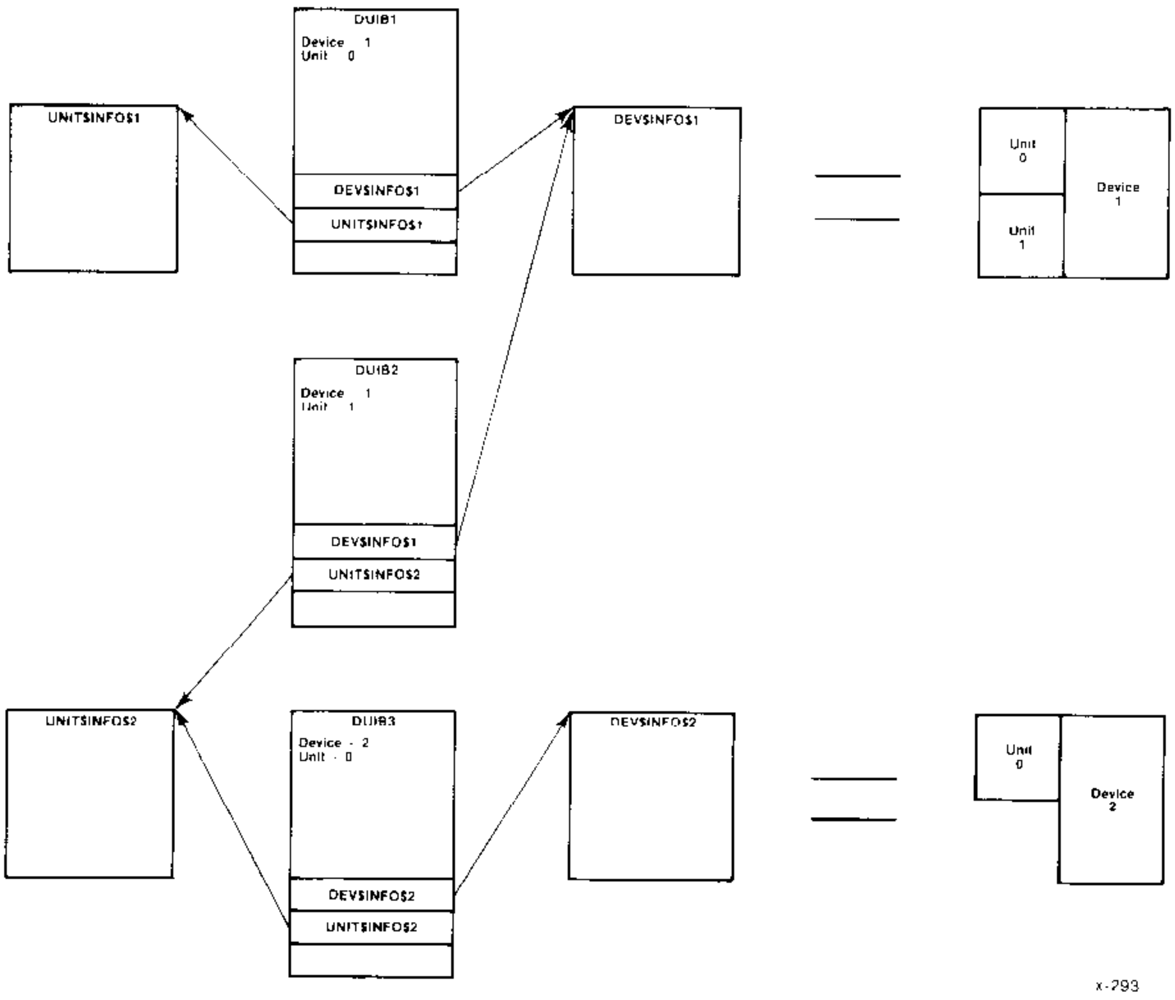


Figure 5-4. DUIBs, Device and Unit Information Tables

5.3.1 Device Information Table

Device Information Tables for common and random access drivers contain the same fields in the same order. If you write your own device drivers, there are two ways to set up Device Information Tables. The first way is to use the User Device Support (UDS) utility to add support for your driver to the ICU. If you use the UDS utility to modify the ICU, you can choose your driver from an ICU menu and fill in the necessary information just as you would when configuring an Intel-supplied driver. Refer to Chapter 9 for more information about the UDS utility.

The second way to set up Device Information Tables is to code them in the format shown here (as assembly-language structures). With this method, you give the ICU the pathname of your Device Information Table file, and the ICU includes the file in the assembly of ?ICDEV.A28, a Basic I/O System configuration file the ICU creates (the ? means the first character of the file can vary). /RMX286/IOS/IDEVCF.INC contains the definition of the structure. For more information on configuring user-written device drivers, see Chapter 9.

The fields DEVICE$INIT, DEVICE$FINISH, DEVICE$START, DEVICE$STOP, and DEVICE$INTERRUPT contain the names of user-supplied procedures whose duties are described later in this chapter. If you decide not to use the UDS utility to add configuration support to the ICU, specify external declarations for these user-supplied procedures when creating the file containing your Device Information Tables. This allows the code for these user-supplied procedures to be included into the assembly of the I/O System. For example, if your procedures are named SAMPLE$DEVICE$INIT, SAMPLE$DEVICE$FINISH, SAMPLE$DEVICE$START, SAMPLE$DEVICE$STOP, and SAMPLE$DEVICE$INTERRUPT, include the following declarations in the file containing your Device Information Tables:

```
extrn sample$device$init: near
extrn sample$device$finish: near
extrn sample$device$start: near
extrn sample$device$stop: near
extrn sample$device$interrupt: near
```

If you use the UDS utility, these external declarations are included automatically.

If you set up your own file of Device Information Tables, use the following format when coding your Device Information Tables:

```
RADEV_DEV_INFO  <
&    LEVEL,               ; WORD
&    PRIORITY,            ; BYTE
&    STACK$SIZE,          ; WORD
&    DATA$SIZE,           ; WORD
&    NUM$UNITS,           ; WORD
&    DEVICE$INIT,         ; WORD
&    DEVICE$FINISH,       ; WORD
&    DEVICE$START,        ; WORD
&    DEVICE$STOP,         ; WORD
&    DEVICE$INTERRUPT,    ; WORD
&    TIMED$OUT,           ; WORD
&    RESERVED$A,          ; WORD
&    RESERVED$B,          ; WORD
&    QUEUE$SIZE,          ; WORD
&    INSTANCE,            ; BYTE
&    BOARD$ID (10)        ; BYTE
&    >
```

where

LEVEL — WORD distinguishing message-passing devices from interrupt-driven devices.

For interrupt-driven devices, this field specifies an encoded interrupt level at which the device will interrupt. The interrupt task uses this value to associate itself with the correct interrupt level. The values for this field are encoded as follows:

Bits	Value
15	If one, this is an extended device information structure. That is, it contains the last three fields (TIMED$OUT, RESERVED$A, and RESERVED$B). If zero, the random access support code doesn't recognize those fields, even if they are present.
14-7	0
6-4	First digit of the interrupt level (0-7).
3	If one, the level is a master level and bits 6-4 specify the entire level number. If zero, the level is a slave level and bits 2-0 specify the second digit.
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero.

For message-passing devices, this field specifies the device as follows:

Bits	Value
15	0
14	Set to one to indicate a message-passing device.
13-8	0
7-0	Specifies the type of message interface. Currently only the value zero is supported.

PRIORITY	For interrupt-driven controllers, a BYTE specifying the initial priority of the interrupt task. The actual priority of an interrupt task might change because the Nucleus adjusts an interrupt task's priority according to the interrupt level it services. Refer to the *Extended iRMX II Nucleus User's Guide* for further information about this relationship between interrupt task priorities and interrupt levels. For message-passing controllers, this value specifies the fixed priority of the task receiving messages from the controller.
STACK$SIZE	WORD specifying the size, in bytes, of the stack for the user-written device interrupt procedure (and procedures that it calls). This number should not include stack requirements for the I/O System-supplied procedures. They add their requirements to this figure.
DATA$SIZE	WORD specifying the size, in bytes, of the user portion of the device's data storage area. This figure should not include the amount needed by the I/O System-supplied procedures; rather, it should include only that amount needed by the user-written routines.
NUM$UNITS	WORD specifying the number of units supported by the driver. Units are assumed to be numbered consecutively, starting with zero.
DEVICE$INIT	WORD specifying the offset address of a user-written device initialization procedure. The format of this procedure, which INIT$IO calls, is described later in this chapter.
DEVICE$FINISH	WORD specifying the offset address of a user-written device finish procedure. The format of this procedure, which FINISH$IO calls, is described later in this chapter.

DEVICE$START	WORD specifying the offset address of a user-written device start procedure. The format of this procedure, which QUEUE$IO and INTERRUPT$TASK/MESSAGE$TASK call, is described later in this chapter.
DEVICE$STOP	For interrupt-driven devices, a WORD specifying the offset address of a user-written device stop procedure. The format of this procedure, which CANCEL$IO calls, is described later in this chapter. For message-passing devices, CANCEL$IO does not call this procedure.
DEVICE$- INTERRUPT	WORD specifying the offset address of a user-written device interrupt procedure. The format of this procedure, which INTERRUPT$TASK/MESSAGE$TASK calls, is described later in this chapter.
TIMED$OUT	For interrupt-driven devices, a WORD specifying the timeout value for the TIMED$INTERRUPT system call. This value represents the number of Nucleus clock intervals the TIMED$INTERRUPT system call waits without receiving an interrupt before it returns with an error. If LEVEL bit 15 is set to zero, the default value for TIMED$OUT will be 0FFFFH, which means the task will wait forever. For message-passing devices, this value specifies the number of Nucleus clock intervals the message task should wait for a message from the controller. If the message task times out without having received a message and I/O requests are pending, the message task tries to receive the message again. If this attempt succeeds, the previous timeout is ignored. If it fails, all pending requests are flushed from the queue with an E$TIME exception. The time random access support may take to return an IORS with this status may vary from the timeout you specify to (timeout * 2). To specify the message task should not timeout (wait forever), specify "0FFFFH."
RESERVED$A RESERVED$B	Intel reserves these WORDs for future use. Set these values to zero.

QUEUE$SIZE	WORD specifying the maximum number of controller messages the Nucleus Communications Service will queue at the port from which the message task receives these messages. Adding one to this value increases this port's memory requirements by five bytes.
INSTANCE	BYTE specifying a particular board in a system containing multiple occurrences of this board name. Boards having the same name are assumed to have instance IDs allocated in contiguous order, starting from an ID of one for the occurrence of the board with the lowest slot id.
BOARD$ID	A BYTE array containing the 10-character board name stored in registers two through eleven of the header record in this board's interconnect space.

Depending on the requirements of your device, you can append additional information to the RADEV_DEV_INFO structure. For example, most devices require you to append the I/O port address to this structure, so that the user-written procedures have access to the device.

5.3.2 Unit Information Table

If you have random access device drivers in your system, you must create a Unit Information Table for each different type of unit in your system. Each random access device-unit's DUIB must point to one Unit Information Table, although multiple DUIBs can point to the same Unit Information Table. The Unit Information Table must include all information that is unit dependent.

As with Device Information Tables, there are two ways to create Unit Information Tables to support user-written device drivers. The first way is to use the User Device Support (UDS) utility to add support for your driver to the ICU. If you use the UDS utility to modify the ICU, you can choose your driver from an ICU menu and fill in the necessary information just as you would when configuring an Intel-supplied driver. Refer to Chapter 9 for more information about the UDS utility.

The second way to set up Unit Information Tables is to code them in the format shown here (as assembly-language structures). If you give the ICU the pathname of your Unit Information Table file, the ICU includes the file in the assembly of ?ICDEV.A28, a Basic I/O System configuration file that the ICU creates (the ? means that the first character of the file name can vary). /RMX286/IOS/IDEVCF.INC contains the definition of the structure. For more information on configuring user-written devices, see Chapter 9.

The minimum requirements for the structure of the Unit Information Table are as follows:

```
RADEV_UNIT_INFO  <
&   TRACK$SIZE,          ; WORD
&   MAX$RETRY,         ; WORD
&   CYLINDER$SIZE      ; WORD
&   >
```

where

TRACK$SIZE	WORD specifying the size, in bytes, of a single track of a volume on the unit. If the device controller supports reading and writing across track boundaries, and your driver is a random-access driver, place a zero in this field. In this case, the I/O System-supplied random access support procedures place an absolute sector number in the DEV$LOC field of the IORS. If you specify a nonzero value for this field, the random access support procedures guarantee that read and write requests do not cross track boundaries. They do this by placing the sector number in the low-order word of the DEV$LOC field of the IORS and the track number in the high-order word of the DEV$LOC field before calling a user-written device start procedure. Instructions for writing a device start procedure are contained later in this chapter.
	For message-passing devices, set this value to zero.
MAX$RETRY	For interrupt-driven devices, a WORD specifying the maximum number of times an I/O request should be tried if an error occurs. Nine is the recommended value for this field. When this field contains a nonzero value, the I/O System-supplied procedures guarantee that read or write requests are retried if the user-supplied device start or device interrupt procedures return an IO$SOFT condition in the IORS.UNIT$STATUS field. (The IORS.UNIT$STATUS field is described in the "IORS Structure" section of Chapter 4.)
	For message-passing devices, this field is ignored.

CYLINDER$SIZE	For interrupt-driven devices, a WORD whose meaning depends on its value, as follows:

0	The random access support code never splits a read or write into a seek/read or a seek/write. Instead, either it expects the device driver to request seek operations whenever a read/write begins on a cylinder different from the one associated with the current position of the read/write head (explicit seeks), or it expects the controller to perform these seeks automatically (implied seeks).
1	The I/O System automatically requests a seek operation (to seek to the correct cylinder) before performing any read or write. The device driver for the unit must call the SEEK$COMPLETE procedure immediately following each seek operation.
Other	Any other value specifies the number of sectors in a cylinder on the unit. The I/O System uses this information to determine when it should request seek operations. It automatically requests a seek operation whenever a requested read or write operation begins in a different cylinder than that associated with the current position of the read/write head. The device driver for the unit must call the SEEK$COMPLETE procedure immediately following each seek operation.

For message-passing devices, this field is ignored.

5.4 DEVICE DATA STORAGE AREA

The common and random access device drivers are set up so that all data that is local to a device is maintained in an area of memory. The INIT$IO procedure creates this device data storage area, and the other procedures of the driver access and update information in it as needed. Storing the device-local data in a central area serves two purposes.

First, all device driver procedures that service individual units of the device can access and update the same data. The INIT$IO procedure passes the address of the area back to the I/O System, which in turn gives the address to the other procedures of the driver.

They can then place information relevant to the device as a whole into the area. The identity of the first IORS on the request queue is maintained in this area, as well as the attachment status of the individual units and a means of accessing the interrupt/message task.

Second, several devices of the same type can share the same device driver code and still maintain separate device data areas. For example, suppose two iSBC 214 devices use the same device driver code. The same INIT$IO procedure is called for each device, and each time it is called it obtains memory for the device data. However, the memory areas that it creates are different. Only the routines that service units of a particular device are able to access the device data area for that device.

Because the common and random access device drivers provide this mechanism, you might also want to include a device data storage area in any custom driver that you write.

5.5 INTRODUCTION TO PROCEDURES DRIVERS MUST SUPPLY

The routines provided by the I/O System and that the I/O System calls (INIT$IO, FINISH$IO, QUEUE$IO, CANCEL$IO, and INTERRUPT$TASK/MESSAGE$TASK) constitute the bulk of a common or random access device driver. These routines, in turn, make calls to device-dependent routines that you (if you are writing your own random access or common driver) must supply. These device-dependent routines are described here briefly and then are presented in detail:

A device initialization procedure. This procedure must perform any initialization functions necessary to get the device ready to process I/O requests. INIT$IO calls this procedure.

A device finish procedure. This procedure must perform any necessary final processing on the device so that the device can be detached. FINISH$IO calls this procedure.

A device start procedure. This procedure must start the device processing any possible I/O function. QUEUE$IO and INTERRUPT$TASK/MESSAGE$TASK (the random access-supplied interrupt/message task) call this procedure.

A device stop procedure. For interrupt-driven drivers, this procedure must stop the device from processing the current I/O function, if that function could take an indefinite amount of time. CANCEL$IO calls the device stop procedure.

For message-passing drivers, all I/O functions are guaranteed to finish within a limited time. CANCEL$IO does not call this procedure; therefore, this procedure does not need to perform any operations.

A device interrupt procedure. This procedure must do all of the device-dependent processing that results from the device's sending an interrupt/message. INTERRUPT$TASK/MESSAGE$TASK calls this procedure.

Figure 5-5 illustrates the relationship between these procedures and the higher-level procedures (INIT$IO, FINISH$IO, QUEUE$IO, and CANCEL$IO) supplied by the I/O System. The remaining sections of this chapter discuss the procedures in detail.

Chapter 8 also provides information about these procedures. Refer to that chapter for specifics on what the procedures must do when handling each type of I/O request.

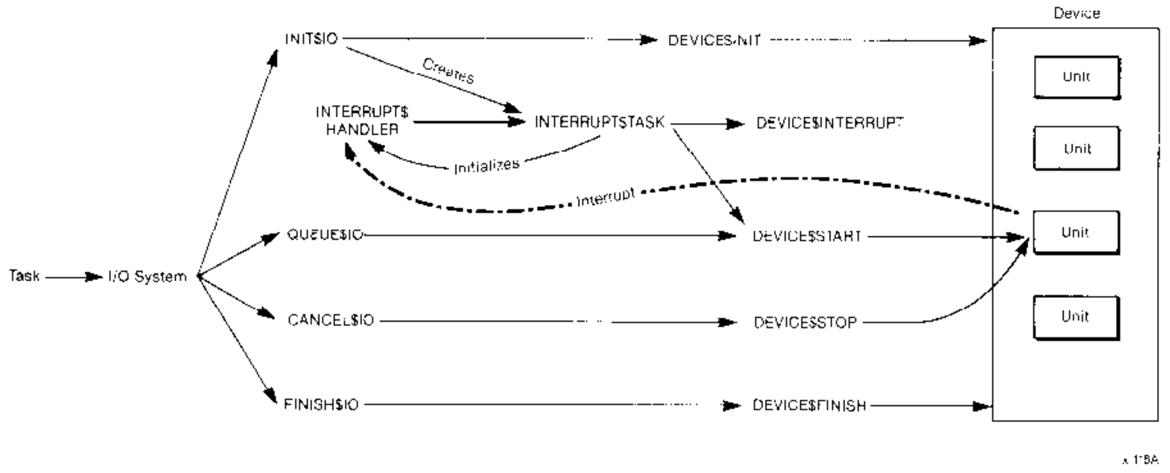


Figure 5-5. Relationships between Random-Access Driver Procedures

5.6 DEVICE INITIALIZATION PROCEDURE

The INIT$IO procedure calls the user-written device initialization procedure to initialize the device. The format of the call to the user-written device initialization procedure is as follows:

```
CALL device$init(duib$p, ddata$p, status$p);
```

where

device$init	Name of the device initialization procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
duib$p	POINTER to the DUIB of the device-unit being attached. INIT$IO supplies this pointer as an input parameter. From this DUIB, the device initialization procedure can obtain the Device Information Table, where information such as the I/O port address is stored.

ddata$p	POINTER to an area of memory supplied by the random access support code. This memory is the user portion of the device's data storage area. You must specify the size of this area of memory in the Device Information Table for this device. The device initialization procedure can use this data area for whatever purposes it chooses. Possible uses for this data area include local flags and buffer areas.

For message-passing devices, this portion of the device's data storage area begins with the following fields:

port$t	TOKEN for the Nucleus Communications Service port where the message task waits for messages from the controller.
slot$id	BYTE indicating the cardslot number for this controller.

status$p	POINTER to a WORD that INIT$IO supplies as an output parameter. The device initialization procedure must return the status of the initialization operation in this word. It should return the E$OK condition code if the initialization is successful. Otherwise, it should return the appropriate exceptional condition code. If initialization does not complete successfully, the device initialization procedure must ensure that any resources it creates are deleted.

The device initialization procedure must do the following:

- It must initialize any driver data structures or flags.

  For message-passing drivers, this procedure must initialize the PORT$T and SLOT$ID fields of the device's data storage area. The device initialization procedure must create a port, then store this TOKEN in the PORT$T field. This procedure must also scan interconnect space for the board instance specified in the Device Information Table and return its slot ID to the SLOT$ID field. Intel has supplied sample code implementing both of these steps on Release Diskette Number 19: iRMX II Demonstration Software that comes with your Extended iRMX II package. For a description of these examples, see the *Extended iRMX II Nucleus User's Guide.*

  If you have a device that does not need to be initialized before it can be used, you can use DEFAULT$INIT, the default device initialization procedure supplied by the I/O System. Specify this name in the Device Information Table. DEFAULT$INIT does nothing but return the E$OK condition code.

- It must reset the board or device. Then it can wait for completion of the reset.

  For interrupt-driven drivers, the device initialization procedure will not receive the interrupt if the device sends an interrupt to indicate the reset is complete. For such devices, either the device start or device interrupt procedure should contain special code to process the reset interrupt.

For message-passing drivers, the device initialization procedure will receive initialization responses from the controller. Either this procedure, the device start, or device interrupt procedure can process such responses.

5.7 DEVICE FINISH PROCEDURE

The FINISH$IO procedure calls the user-written device finish procedure to perform final processing on the device, after the last I/O request has been processed. The format of the call to the device finish procedure is as follows:

```
CALL device$finish(duib$p, ddata$p);
```

where

device$finish	Name of the device finish procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
duib$p	POINTER to the DUIB of the device-unit being detached. FINISH$IO supplies this parameter as an input parameter. From this DUIB, the device finish procedure can obtain the Device Information Table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. This is an input parameter supplied by FINISH$IO. The device finish procedure should obtain, from this data area, identification of any resources other user-written procedures may have created, and delete these resources.

If you have a device that does not require any final processing, you can use the default device finish procedure supplied by the I/O System. The name of this procedure is DEFAULT$FINISH. Specify this name in the Device Information Table. DEFAULT$FINISH merely returns control to the caller. It is normally used when the default initialization procedure DEFAULT$INIT is used.

5.8 DEVICE START PROCEDURE

QUEUE$IO and INTERRUPT$TASK/MESSAGE$TASK make calls to the device start procedure to start an I/O function. QUEUE$IO calls this procedure on receiving an I/O request when the request queue is empty. INTERRUPT$TASK/MESSAGE$TASK calls the device start procedure after it finishes one I/O request if there are one or more I/O requests on the queue. The format of the call to the device start procedure is as follows:

```
CALL device$start(iors$p, duib$p, ddata$p);
```

where

device$start	Name of the device start procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
iors$p	POINTER to the IORS of the request. This is an input parameter supplied by QUEUE$IO or INTERRUPT$TASK/MESSAGE$TASK. The device start procedure must access the IORS to obtain information such as the type of I/O function requested, the address on the device of the block (absolute sector) where I/O is to commence, and the buffer address.
duib$p	POINTER to the DUIB of the device-unit for which the I/O request is intended. This is an input parameter supplied by QUEUE$IO or INTERRUPT$TASK/MESSAGE$TASK. The device start procedure can use the DUIB to access the Device Information Table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. This is an input parameter supplied by QUEUE$IO or INTERRUPT$TASK/MESSAGE$TASK. The device start procedure can use this data area to set flags or store data.

The device start procedure must do the following:

- It must be able to start the device processing any of the I/O requests supported by the device and recognize that requests for nonsupported functions are error conditions. It must process the I/O requests as described in Chapter 8.
- If it transfers any data, it must update the IORS.ACTUAL field to reflect the total number of bytes of data transferred (that is, if it transfers 128 bytes of data, it must put 128 in the IORS.ACTUAL field).
- If an error occurs when the device start procedure tries to start the device (such as on a write request to a write-protected disk), the device start procedure must set the IORS.STATUS field to indicate an E$IO condition and the IORS.UNIT$STATUS field to a nonzero value. The lower four bits of the IORS.UNIT$STATUS field should be set as indicated in the "IORS Structure" section of Chapter 4. The remaining bits of the field can be set to any value (some device drivers return the device's result byte in the remainder of this field). If the function completes without an error, the device start procedure must set the IORS.STATUS field to indicate an E$OK condition.
- **For message-passing devices**, the device start procedure must set the IORS.DONE field to any **even** value between TRUE (0FFH) and FALSE (0H) if the request has been started and is in progress.

  If the device start procedure determines that the I/O request has been processed completely, either because of an error or because the request has completed successfully, it must set the IORS.DONE field to TRUE. The I/O request will not always be completed; it may take several calls to the device interrupt procedure before a request is completed. However, if the request is finished and the device start procedure does not set the IORS.DONE field to TRUE, the random access support routines wait until the device sends an interrupt/message indicating the request is finished and the device interrupt procedure sets IORS.DONE to TRUE, before determining that the request is actually finished.

5.9 DEVICE STOP PROCEDURE

For interrupt-driven devices, the CANCEL$IO procedure calls the user-written device stop procedure to stop the device from performing the current I/O function. The format of the call to the device stop procedure is as follows:

```
CALL device$stop(iors$p, duib$p, ddata$p);
```

where

device$stop	Name of the device stop procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include this name in the Device Information Table.
iors$p	POINTER to the IORS of the request. This is an input parameter supplied by CANCEL$IO. The device stop procedure needs this information to determine what type of function to stop.
duib$p	POINTER to the DUIB of the device-unit on which the I/O function is being performed. This is an input parameter supplied by CANCEL$IO.
ddata$p	POINTER to the user portion of the device's data storage area. This is an input parameter supplied by CANCEL$IO. The device stop procedure can use this area to store data, if necessary.

If you have a device which guarantees that all I/O requests will finish in an acceptable amount of time, you can omit writing a device stop procedure (message-passing devices are such devices). Instead, use DEFAULT$STOP, the default procedure supplied with the I/O System. Specify this name in the Device Information Table. DEFAULT$STOP simply returns to the caller.

5.10 DEVICE INTERRUPT PROCEDURE

INTERRUPT$TASK/MESSAGE$TASK calls the user-written device interrupt procedure to process an interrupt/message that just occurred.

For interrupt-driven devices, the format of the call to the device interrupt procedure is as follows:

```
CALL device$interrupt(iors$p, duib$p, ddata$p);
```

where

device$interrupt	Name of the device interrupt procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include this name in the Device Information Table.
iors$p	POINTER to the IORS of the request being processed. This is an input parameter supplied by INTERRUPT$TASK. The device interrupt procedure must update information in this IORS. A value of NIL for this parameter indicates either that there are no requests on the request queue and the interrupt is extraneous or that the unit is completing a seek or other long-term operation.
duib$p	POINTER to the DUIB of the device-unit on which the I/O function was performed. This is an input parameter supplied by INTERRUPT$TASK.
ddata$p	POINTER to the user portion of the device's data storage area. This is an input parameter supplied by INTERRUPT$TASK. The device interrupt procedure can update flags in this data area or retrieve data sent by the device.

For message-passing devices, the format to the call to the device interrupt procedure is as follows:

```
CALL device$interrupt(message$p, ddata$p, status$p);
```

where

device$interrupt	Name of the device interrupt procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include this name in the Device Information Table.

message$p — A POINTER to a STRUCTURE of the following type:

```
STRUCTURE(
        data$p          POINTER,
        flags           WORD,
        status          WORD,
        trans$id        WORD,
        data$length     DWORD,
        dummy1          WORD,
        socket          DWORD,
        control(20)     BYTE,
        dummy2(12)      BYTE);
```

where

data$p is a POINTER to the data message received. If the data was received in a data chain, this is a pointer to the data chain. If the pointer value is NIL, then a control message containing no data was received.

flags is a WORD with the following encoded meaning:

Bit	Name
0	data$type
1-2	receive$type
3-15	reserved

where

data$type defines whether data$ptr points to a data chain (1B) or a single buffer (0B.)

receive$type is an indicator of the type of message received as follows:

Value	Message Type
00B	Transactionless message (RQ$SEND or similar call)
01B	Transmission or system status message
10B	Transaction request message (RQ$SEND$RSVP or similar call)
11B	Transaction response message (RQ$SEND$REPLY or similar call)

status contains the send message status returned by the Nucleus Communications Service. The status codes are

Status	Meaning
E$OK	A new message has been successfully received
E$CANCELED	A SEND$RSVP transaction has been remotely canceled.
E$NO$LOCAL$-BUFFER	This error applies to two cases: If the receive$type parameter indicates a request message, the local port's buffer pool does not contain a buffer large enough to hold the message so the RQ$RECEIVE$FRAGMENT system call is required (message fragmentation.) If the receive$type parameter indicates a response message, the RSVP buffer supplied in the RQ$SEND$RSVP system call is not large enough to hold the response.
E$NO$REMOTE$-BUFFER	The remote port's buffer pool does not have a buffer large enough to hold the message and message fragmentation is turned off.
E$TRANSMISSION	A NACK (Negative Acknowledgment), MPC Failsafe timeout, bus or agent error, or retry expiration occurred during the transmission of the message.

trans$id is a WORD that contains the transaction ID for this message. If a transactionless message was received, trans$id is invalid. The device interrupt procedure must map trans$id to the correct IORS. To do this mapping the driver must also maintain a queue of all started requests along with their matching transaction IDs.

data$length is a DWORD that indicates the length of the data message received.

If **receive$type** indicates a newly received message, then data$length contains the length of the successfully received message.

If **receive$type** and **status** indicate request message fragmentation, the data$length contains the length of all the message fragments that will be received using the RQ$RECEIVE$FRAGMENT system call.

dummy1 is a WORD Intel reserves for future use. Set this value to zero.

socket is a DWORD containing the host$id:port$id that indicates the message source.

control is the 20-byte long control part of a data message.

dummy2 is a reserved BYTE array. Set to zero.

ddata$p	POINTER to the user portion of the device's data storage area. This is an input parameter supplied by MESSAGE$TASK. The device interrupt procedure can update flags in this data area or retrieve data sent by the device.
status$p	POINTER to a WORD containing the device status code returned by the user-supplied device interrupt procedure. This parameter should return an E$OK condition unless a board failure occurs.

The device interrupt procedure must do the following:

- **For interrupt-driven devices**, it must determine from the IORS it receives which unit sent the interrupt and what action to take. **For message-passing devices**, it must determine this information from the data message received.
- After determining the device-unit, the device interrupt procedure must decide whether the request is finished. If the request is finished, the device interrupt procedure must set the IORS.DONE field to TRUE.
- It must process the interrupt/message. This may involve setting flags in the user portion of the data storage area, transferring data written by the device to a buffer, or some other operation.
- If an error has occurred, it must set the IORS.STATUS field to indicate an E$IO condition and the IORS.UNIT$STATUS field to a nonzero value. The lower four bits of the IORS.UNIT$STATUS field should be set as indicated in the "IORS Structure" section of Chapter 4. The remaining bits of the field can be set to any value (some device drivers return the device's result byte in the remainder of this field). It must

also set the IORS.DONE field to TRUE, indicating that the request is finished because of the error.

For message-passing drivers, STATUS$P returns an error only if an unrecoverable controller failure occurs. MESSAGE$TASK will mark all pending IORSs DONE with their status set to the error returned by STATUS$P, then flush them from the request queue.

- If no error has occurred, it must set the IORS.STATUS field to indicate an E$OK condition.

5.11 PROCEDURES RANDOM ACCESS DRIVERS MUST CALL

There are several procedures that random access drivers must call under certain well-defined circumstances. They are NOTIFY, SEEK$COMPLETE, and procedures for the long-term operations (BEGIN$LONG$TERM$OP, END$LONG$TERM$OP, and GET$IORS).

5.11.1 NOTIFY Procedure

Whenever a door to a flexible diskette drive is opened or the STOP button on a hard disk drive is pressed, the device driver for that device must notify the I/O System that the device is no longer available. The device driver does this by calling the NOTIFY procedure. When called in this manner, the I/O System stops accepting I/O requests for files on that device unit. Before the device unit can again be available for I/O requests, the application must detach it by a call to A$PHYSICAL$DETACH$DEVICE and reattach it by a call to A$PHYSICAL$ATTACH$DEVICE. Moreover, the application must obtain new file connections for files on the device unit.

In addition to not accepting I/O requests for files on that device unit, the I/O System will respond by sending an object to a mailbox. For this to happen, however, the object and the mailbox must have been established for this purpose by a prior call to A$SPECIAL, with the spec$func argument equal to FS$NOTIFY (2). (The A$SPECIAL system call is described in the *Extended iRMX II Basic I/O System Calls* manual.) The task that awaits the object at the mailbox has the responsibility of detaching and reattaching the device unit and of creating new file connections for files on the device unit.

The syntax of the NOTIFY procedure is as follows:

```
CALL NOTIFY(unit, ddata$p);
```

where

unit	BYTE containing the unit number of the unit on the device that went off-line.

ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that is passed to the device driver by way of either the device$start or the device$interrupt procedure.

5.11.2 SEEK$COMPLETE Procedure

In most applications, it is desirable to overlap seek operations (which can take relatively long periods of time) with other operations on other units of the same device. To facilitate this, a device driver receiving a seek request can take the following actions in the following order:

1. The device start procedure starts the requested seek operation.
2. Depending on the kind of device, either the device start procedure or the device interrupt procedure sets the DONE flag in the IORS to TRUE (0FFH).
    - Some devices send only one interrupt/message in response to a seek request--the one that indicates the completion of the seek. If your device operates in this manner, the device start procedure sets the DONE flag to TRUE (0FFH) immediately.
    - Some devices send two interrupts/messages in response to a seek request--one upon receipt of the request and one upon completion of the seek. If your device operates in this manner, the device start procedure leaves the DONE flag in the IORS set to FALSE (0).

      When the first interrupt/message from the device arrives, the device interrupt procedure sets the DONE flag to TRUE (0FFH).
3. When the interrupt/message from the device arrives (the one that indicates the completion of the seek), the device interrupt procedure calls the SEEK$COMPLETE procedure to signal the completion of the seek operation.

This process enables the device driver to handle I/O requests for other units on the device while the seek is in progress, thereby increasing the performance of the I/O System.

The syntax of the call to SEEK$COMPLETE is as follows:

```
CALL SEEK$COMPLETE(unit, ddata$p);
```

where

unit	BYTE containing the number of the unit on the device on which the seek operation is completed.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines passes to the device start and device interrupt procedures.

Note that if your device driver calls the SEEK$COMPLETE procedure when a seek operation is completed, the CYLINDER$SIZE field of the Unit Information Table for the device unit should be configured greater than zero. On the other hand, if you configure CYLINDER$SIZE to zero (indicating that you don't want to overlap seek operations), your driver should never call SEEK$COMPLETE.

5.11.3 Procedures for Other Long-Term Operations

The Operating System provides three procedures which device drivers can use to overlap long-term operations (such as tape rewinds) with other I/O operations. The procedures are BEGIN$LONG$TERM$OP, END$LONG$TERM$OP, and GET$IORS. These procedures are intended specifically for use with devices that do not support seek operations (such as tape drives).

5.11.3.1 BEGIN$LONG$TERM$OP Procedure

The BEGIN$LONG$TERM$OP procedure informs the random access support routines that a long-term operation is in progress, and that the support routines do not have to wait for the operation to complete before servicing other units on the device. Calling BEGIN$LONG$TERM$OP allows the controller to service read and write requests on other units of the device while the long-term operation is in progress.

To use BEGIN$LONG$TERM$OP, the device driver receiving the request for the long-term operation should take the following actions:

1. The device start procedure starts the long-term operation.
2. Depending on the kind of device, either the device start procedure or the device interrupt procedure sets the DONE flag in the IORS to TRUE (0FFH).
   - Some devices send only one interrupt/message in response to a request for a long-term operation--the one that indicates the completion of the operation. If your device operates in this manner, the device start procedure sets the DONE flag to TRUE (0FFH) immediately.
   - Some devices send two interrupt/messages in response to a request for a long-term operation--one upon receipt of the request and one upon completion of the operation. If your device operates in this manner, the device start procedure leaves the DONE flag in the IORS set to FALSE (0). When the first interrupt/message from the device arrives, the device interrupt procedure sets the DONE flag to TRUE (0FFH).
3. The procedure that just set the DONE flag to TRUE (either the device start or device interrupt procedure) calls BEGIN$LONG$TERM$OP.

The syntax of the call to BEGIN$LONG$TERM$OP is as follows:

```
CALL BEGIN$LONG$TERM$OP(unit, ddata$p);
```

where

unit	BYTE containing the number of the unit on the device that is performing the long-term operation.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines pass to the device start and device interrupt procedures.

If your driver calls BEGIN$LONG$TERM$OP, it must also call END$LONG$TERM$OP when the device sends an interrupt/message to indicate the end of the long-term operation.

5.11.3.2 END$LONG$TERM$OP Procedure

The END$LONG$TERM$OP procedure informs the random access support routines that a long-term operation has completed. A driver that calls BEGIN$LONG$TERM$OP must also call END$LONG$TERM$OP or the driver cannot further access the unit that performed the long-term operation.

Specifically, when the unit sends an interrupt/message indicating the end of the long-term operation, the device interrupt procedure must call END$LONG$TERM$OP.

The syntax of the call to END$LONG$TERM$OP is as follows:

```
CALL END$LONG$TERM$OP(unit, ddata$p);
```

where

unit	BYTE containing the number of the unit on the device that performed the long-term operation.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines pass to the device start and device interrupt procedures.

5.11.3.3 GET$IORS Procedure

Long-term operations on some units involve multiple operations. For example, performing a rewind on some tape drives requires you to perform a rewind and a read file mark. The GET$IORS procedure allows your driver procedures to handle this situation without forcing you to write a custom driver for each device that is different.

GET$IORS allows your driver procedure to obtain the token of the IORS for the previous long-term request, so that it can modify the IORS to initiate new I/O requests. The driver cannot access the IORS without calling this procedure, because when the long-term operation completes (and an interrupt/message occurs), the IORS$P that INTERRUPT$TASK passes to the device interrupt procedure is set to zero (for units busy performing a seek or other long-term operation).

To use GET$IORS, the device driver performing the long-term operation should take the following actions:

1. The device driver starts the long-term operation and calls BEGIN$LONG$TERM$OP in the usual manner (as described in the "BEGIN$LONG$TERM$OP Procedure" section).
2. When the unit sends an interrupt/message indicating the end of the long-term operation, the device interrupt procedure calls GET$IORS to obtain the IORS.
3. The device interrupt procedure modifies the FUNCT and SUBFUNCT fields of the IORS to specify the next operation to perform. It also sets the DONE flag to FALSE (0H).
4. The device interrupt procedure calls END$LONG$TERM$OP.

The syntax of the call to GET$IORS is as follows:

```
iors$base = GET$IORS(unit, ddata$p);
```

where

iors$base	SELECTOR in which the random access support routines return the base descriptor of the IORS. Use the PL/M-286 built-in procedure BUILD$PTR (specifying an offset of 0) to obtain a pointer to the IORS.
unit	BYTE containing the number of the unit on the device which performed the long-term operation.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines pass to the device start and device interrupt procedures.

5.12 FORMATTING CONSIDERATIONS

If you write a random access driver and you intend to use the Human Interface FORMAT command to format volumes on that device, your driver routines must set the status field in the IORS in the manner that the FORMAT command expects.

When formatting volumes, the FORMAT command issues system calls (A$SPECIAL or S$SPECIAL) to format each track. It knows that formatting is complete when it receives an E$SPACE exception code in response. To be compatible with FORMAT, your driver must also return E$SPACE when formatting is complete.

In particular, if your driver must perform some operation on the device to format it, your device interrupt procedure must set the IORS.STATUS to E$SPACE after the last track has been formatted.

However, if the device requires no physical formatting (for example, when formatting is a null operation for that device), your device start procedure can set IORS.STATUS to E$SPACE immediately after being called to start the formatting operation.

The Human Interface FORMAT command can report the assignment of alternate tracks, or, if no alternate tracks are available, to mark all the sectors in the track being formatted as unavailable via the bad block map. This enables you to see the state of the media in question and allows a disk with excess bad tracks (more than the available alternate tracks can handle) to continue being used. For the FORMAT command to provide these features, the random access driver must return these error codes in the following conditions:

- Whenever the device driver is processing an F$SPECIAL (FS$FORMAT) command and it allocates an alternate track, it must return an E$IO$ALT$ASSIGNED error code in the IORS after marking the request DONE.
- Whenever the device driver is processing an F$SPECIAL (FS$FORMAT) command and discovers the track is bad, but no alternate tracks are available for assignment, it must return an E$IO$NO$SPARES error code in the IORS after marking the request DONE.

CHAPTER 6 WRITING TERMINAL DRIVERS

The Basic I/O System provides an interface allowing tasks to use the power and convenience of I/O System calls when communicating with terminals. To add support for interrupt-driven or message-passing terminal controllers for which Intel has not supplied device drivers, you can write your own device drivers to provide the software link between the Operating System software (called the Terminal Support Code) and the terminal.

This chapter explains how to write an interrupt-driven or message-passing terminal driver whose capabilities include handling single-character or block-mode I/O, parity checking, answering and hanging up functions on a modem, and automatic baud rate recognition for each of several terminals. It describes the data structures used by terminal drivers, as well as the procedures you must provide.

Throughout this chapter, differences between interrupt-driven and message-passing devices are noted by the following conventions. Message-passing data structures and parameter descriptions are shaded in gray where they differ from interrupt-driven data structures and parameter descriptions. Sections in which no distinction is made apply to both systems. The terms **interrupt** and **message** (and variations of these terms) are noted as "interrupt/message" where they mean that interrupt-driven devices use an interrupt while message-passing devices use a message to accomplish the same purpose. Likewise, the terms **port** and **mailbox** appear as "port/mailbox" where they mean MULTIBUS II drivers use a port to pass messages while MULTIBUS I interruptless drivers use mailboxes.

6.1 TERMINAL SUPPORT CODE

As in the case of common and random-access drivers, the I/O System provides the highest-level driver procedures that the I/O System invokes when performing terminal I/O. These procedures are known collectively as the Terminal Support Code. Figure 6-1 shows schematically the relationships between the various layers of code that are involved in driving a terminal.

The Terminal Support Code supports interrupt-driven and message-passing terminal drivers. It distinguishes between these drivers through the Device Information Table described in this chapter. Among the duties performed by the Terminal Support Code are managing buffers and maintaining several terminal-related modes.

6.2 DATA STRUCTURES SUPPORTING TERMINAL I/O

The principal data structures supporting terminal I/O are the Device-Unit Information Block (DUIB), Device Information Table, Unit Information Table, and the Terminal Support Code (TSC) data structure. These data structures are defined in the next few paragraphs.

If you write your own device drivers, there are two ways to create these structures. The first way is to use the User Device Support (UDS) utility to add support for your driver to the Interactive Configuration Utility (ICU). If you use the UDS utility to modify the ICU, you can choose your driver from an ICU menu and fill in the necessary information just as you would when configuring an Intel-supplied driver. Refer to Chapter 9 for more information about the UDS utility.

The second way to set up these structures is to code them in the format shown in this chapter (as assembly-language structures). With this method, you give the ICU the pathname of the files containing these structures, and the ICU includes the files in the assembly of ?ITDEV.A28 and ?ICDEV.A28, two Basic I/O System configuration files the ICU creates (the ? means that the first character of the file name can vary). /RMX286/IOS/IDEVCF.INC contains the definition of the structures described in this chapter. For more information on configuring user-written device drivers, see Chapter 9.

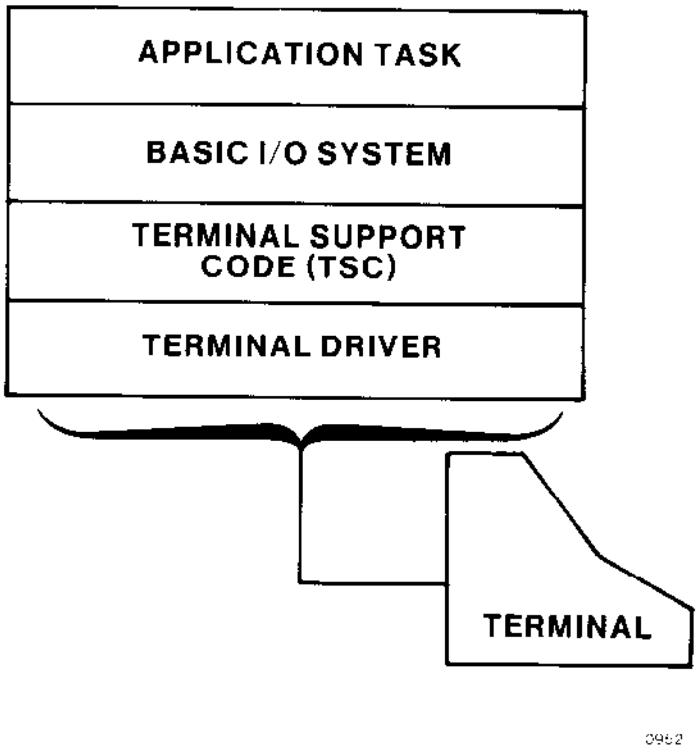


Figure 6-1. Software Layers Supporting Terminal I/O

6.2.1 DUIB

This section lists the elements that make up a DUIB for a device-unit that is a terminal. If you decide not to use the UDS utility to add support to the ICU, code your DUIBs in the format shown here (as assembly-language structures). The numbers and uppercase words indicate items you should enter exactly as shown. The lower case words indicate variable items. Refer to Chapter 4 for more information about the DUIB.

If you give the ICU the pathname of your DUIB file, the ICU includes your DUIB file in the assembly of ?ICDEV.A28, a Basic I/O System configuration file the ICU creates (the ? means that the first character of the name can vary). /RMX286/IOS/IDEVCF.INC contains the definition of the DUIB structure.

```
DEFINE DUIB  <
&   name,                 ; BYTE (14)
&   1,                    ; WORD - file$drivers - (physical)
&   OFBH,                 ; BYTE - functs - (no seek)
&   0,                    ; BYTE - flags - (not disk)
&   0,                    ; WORD - dev$gran - (not random access)
&   0,                    ; DWORD - dev$size - (not storage device)
&   device,               ; BYTE - (device dependent)
&   unit,                 ; BYTE - (unit dependent)
&   dev$unit,             ; WORD - (device and unit dependent)
&   TSINITIO,             ; WORD - init$io - (terminal device)
&   TSFINISHIO,           ; WORD - finish$io - (terminal device)
&   TSQUEUEIO,            ; WORD - queue$io - (terminal device)
&   TSCANCELIO,           ; WORD - cancel$io - (terminal device)
&   device$info$p,        ; POINTER - (address of
&                         ; TERMINAL$DEVICE$INFO)
&   unit$info$p,          ; POINTER - (address of
&                         ; TERMINAL$UNIT$INFO)
&   OFFFFH,               ; WORD - update$timeout - (not disk)
&   0,                    ; WORD - num$buffers - (none)
&   priority,             ; BYTE - (I/O System dependent)
&   0,                    ; BYTE - fixed$update - (none)
&   0,                    ; BYTE - max$buffers - (none)
&   RESERVED,             ; BYTE
&   >
```

6.2.2 Device Information Table

A terminal's Device Information Table provides information about a terminal controller. If you decide not to use the UDS utility to add support to the ICU, code these tables in the format shown here (as assembly-language declarations). If you give the ICU the pathname of your Device Information Table file, the ICU includes the file in the assembly of ?ITDEV.A28, a Basic I/O System configuration file the ICU creates (the ? means that the first character of the name can vary).

The fields TERM$INIT, TERM$FINISH, TERM$SETUP, TERM$OUT, TERM$ANSWER, TERM$HANGUP, TERM$UTILITY, and TERM$CHECK contain the names of user-supplied procedures whose duties are described later in this chapter. When creating the file containing your Device Information Tables, specify external declarations for these user-supplied procedures. This allows the code for these user-supplied procedures to be included in the generation of the I/O System. For example, if your procedures are named SAMPLE$TERM$INIT, SAMPLE$TERM$FINISH, SAMPLE$TERM$SETUP, SAMPLE$TERM$OUT, SAMPLE$TERM$ANSWER, SAMPLE$TERM$HANGUP, SAMPLE$TERM$UTILITY, and SAMPLE$TERM$CHECK, include the following declarations in the file containing your Device Information Tables:

```
extrn sample$term$init: near
extrn sample$term$finish: near
extrn sample$term$setup: near
extrn sample$term$out: near
extrn sample$term$answer: near
extrn sample$term$hangup: near
extrn sample$term$utility: near
extrn sample$term$check: near
```

If you use the UDS utility to add support for your driver to the ICU, these declarations are added automatically.

If you set up your own file of Device Information Tables for interrupt-driven devices, use this format when coding your Device Information Tables:

```
TERMINAL$DEVICE$INFORMATION
    DW  NUM$UNITS
    DW  DRIVER$DATA$SIZE
    DW  STACK$SIZE
    DW  TERM$INIT
    DW  TERM$FINISH
    DW  TERM$SETUP
    DW  TERM$OUT
    DW  TERM$ANSWER
    DW  TERM$HANGUP
    DW  TERM$UTILITY
    DW  NUM$INTERRUPTS

        INTERRUPTS
            DW  INTERRUPT$LEVEL
            DW  TERM$CHECK
                .               ; define interrupt$level and
                .               ; term$check for each interrupt
                .               ; level
        DRIVER$INFO
            DB  DRIVER$INFO$1
            DB  DRIVER$INFO$2
                .
                .
                .
```

If you set up your own file of Device Information Tables for message-passing devices, use this format when coding your Device Information Tables:

```
TERMINAL$DEVICE$INFORMATION
    DW  NUM$UNITS
    DW  DRIVER$DATA$SIZE
    DW  STACK$SIZE
    DW  TERM$INIT
    DW  TERM$FINISH
    DW  TERM$SETUP
    DW  TERM$OUT
    DW  TERM$ANSWER
    DW  TERM$HANGUP
    DW  TERM$UTILITY
    DW  NUM$INTERRUPTS
    DW  TERM$CHECK
    DW  PRIORITY
    DW  RESERVED1
    DW  RESERVED2

DRIVER$INFO
        DB  DRIVER$INFO$1
        DB  DRIVER$INFO$2
                .
                .
                .
```

where

NUM$UNITS	A WORD containing the number of terminals on this terminal controller.
DRIVER$DATA$-SIZE	A WORD containing the number of bytes in the driver's data area pointed to by the USER$DATA$PTR field of the TSC data structure.
STACK$SIZE	A WORD containing the number of bytes of stack needed collectively by the user-supplied procedures in this device driver.
TERM$INIT	A WORD specifying the offset portion of the address of this controller's user-written terminal initialization procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$FINISH	A WORD specifying the offset portion of the address of this controller's user-written terminal finish procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$SETUP	A WORD specifying the offset portion of the address of this controller's user-written terminal setup procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.

TERM$OUT	A WORD specifying the offset portion of the address of this controller's user-written terminal output procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$ANSWER	A WORD specifying the offset portion of the address of this controller's user-written terminal answer procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$HANGUP	A WORD specifying the offset portion of the address of this controller's user-written terminal hangup procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$UTILITY	A WORD specifying the offset portion of the address of this controller's user-written terminal utility procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
NUM$-INTERRUPTS	A WORD by which the Terminal Support Code determines whether this is an interrupt-driven or message-passing device. For interrupt-driven drivers, this word contains the number of interrupt lines this controller uses. You must define an INTERRUPT$LEVEL and TERM$CHECK word for each interrupt. For message-passing drivers, this word must be set to zero.
INTERRUPT$-LEVEL	For interrupt-driven drivers, WORDs containing the encoded level numbers of the interrupts associated with the terminals driven by this controller. You must supply one such word for each interrupt the controller uses. For message-passing drivers, this parameter is not present.

TERM$CHECK	For interrupt-driven drivers, WORDs specifying the offset portions of the addresses of this controller's user-written terminal check procedures. Each TERM$CHECK field specifies the terminal check procedure for the INTERRUPT$LEVEL immediately preceding it. When creating the Device Information Table, use the procedure names as the variables to supply this information. If any of the TERM$CHECK words equals zero, there is no TERM$CHECK procedure associated with the corresponding interrupt level. Instead, interrupts on these levels are assumed to be output ready interrupts which will cause TERM$OUT to be called. For message-passing drivers, a WORD specifying the offset portion of the address of this controller's user-written terminal check procedure. Only one TERM$CHECK procedure is valid. When creating the Device Information Table, use the procedure name as a variable to supply this information.
PRIORITY	For interrupt-driven drivers, this parameter is not present. For message-passing drivers, a WORD specifying the priority of the Terminal Support Code's message task. This task receives messages from the controller.
RESERVED1 RESERVED2	For message-passing drivers, WORDs Intel reserves for future use. Set these words to zero.
DRIVER$INFO	BYTEs or WORDs containing driver-dependent information.

NOTE

Usually, terminal drivers are concerned only with the DRIVER$INFO fields of the Device Information Table. Therefore, a terminal driver can declare a structure of the following form when accessing this data:

```
DECLARE
    TERMINAL$DEVICE$INFO STRUCTURE(
    FILLER(nbr$of$words)    WORD,
    DRIVER$INFO$1           BYTE,
    DRIVER$INFO$2           BYTE,
          .
          .
          .
    DRIVER$INFO$N           BYTE);
```

where

nbr$of$words equals 11 + [2 * (number of interrupt levels used by the driver)] for interrupt-driven drivers.

For message-passing drivers, set this value to 15.

You must supply the terminal initialization, terminal finish, terminal setup, terminal output, terminal answer, terminal hangup, terminal utility, and terminal check procedures. However, if your terminals are not used with modems, you can use the I/O System-supplied TERM$NULL procedure instead of writing your own terminal answer and terminal hangup procedures. Also, you can use TERM$NULL in place of the terminal utility procedure if your terminal is not a buffered device. Finally, you can use TERM$NULL in place of the terminal finish procedure if your application does not need to perform special processing when the last terminal on the controller is detached. TERM$NULL merely returns control to the caller. To use this procedure, specify its name in the Device Information Table.

6.2.3 Unit Information Table

A terminal's Unit Information Table provides information about an individual terminal. Although only one Device Information Table can exist for each driver (controller), several Unit Information Tables can exist if different terminals have different characteristics (such as baud rate, duplex, or parity, for example). If you decide not to use the UDS utility to add support to the ICU, code the Unit Information Tables in the format shown here (as assembly-language declarations). If you give the ICU the pathname of your Unit Information Table field, the ICU includes the file in the assembly of ?ITDEV.A28, a Basic I/O System configuration file the ICU creates (the "?" means the first character of the name can vary).

```
PUBLIC  TERMINAL$UNIT$INFORMATION
    DW  CONN$FLAGS
    DW  TERMINAL$FLAGS
    DW  IN$RATE
    DW  OUT$RATE
    DW  SCROLL$NUMBER
            .
            .
            .
```

where

CONN$FLAGS WORD specifying the default connection flags for this terminal. Refer to the description of the A$SPECIAL system call in the *Extended iRMX II Basic I/O System Calls* manual for more information about these flags. The flags are encoded as follows (bit 0 is the low-order bit):

Bits	Value and Meaning
0-1	Line editing control.
	0 = Invalid Entry.
	1 = No line editing (transparent mode).
	2 = Line editing (normal mode).
	3 = No linc editing (flush mode).
2	Echo control.
	0 = Echo.
	1 = Do not echo.
3	Input parity control.
	0 = Terminal Support Code sets parity bit to 0.
	1 = Terminal Support Code does not alter parity bit.
4	Output parity control.
	0 = Terminal Support Code sets parity bit to 0.
	1 = Terminal Support Code does not alter parity bit.
5	Output control character control.
	0 = Accept output control characters in the input stream.
	1 = Ignore output control characters in the input stream.
6-7	OSC control sequence control.
	0 = Act upon OSC sequences that appear in either the input or output stream.
	1 = Act upon OSC sequences in the input stream only.
	2 = Act upon OSC sequences in the output stream only.

	Bits	Value and Meaning
		3 = Do not act upon any OSC sequences.
	8-15	Reserved bits. For future compatibility, set to 0.

TERMINAL$-FLAGS — WORD specifying the terminal connection flags for this terminal. Refer to the description of the A$SPECIAL system call in the *Extended iRMX II Basic I/O System Calls* manual for more information about these flags. The flags are encoded as follows (bit 0 is the low-order bit):

Bits	Value and Meaning
0	Reserved bit. Set to 1.
1	Line protocol indicator. 0 = Full duplex. 1 = Half duplex.
2	Output medium. 0 = Video display terminal (VDT). 1 = Printed (Hard copy).
3	Modem indicator. 0 = Not used with a modem. 1 = Used with a modem.
4-5	Input parity control. For devices that support link parameters, the LINK$PARAMETER field (when enabled) overrides this parity setting. 0 = Driver always sets parity bit to 0. This yields 8-bit data. This is true even if the LINK$PARAMETER is enabled. 1 = Driver never alters the parity bit. This yields 8-bit data.

Bits	Value and Meaning
	2 = Driver expects even parity on input. This yields 7-bit data. Except for the Terminal Communications Controller driver, if an error occurs, the driver sets the eighth bit to one. Errors include (a) a parity error, (b) the received stop bit has a value of 0 (framing error) or (c) the previous character received has not yet been fully processed (overrun error).
	3 = Driver expects odd parity on input. This yields 7-bit data. Except for the Terminal Communications Controller driver, if an error occurs, the driver sets the eighth bit to one. Errors include (a) a parity error, (b) the received stop bit has a value of 0 (framing error) or (c) the previous character received has not yet been fully processed (overrun error). For the Terminal Communications Controller driver, if a parity error occurs, the character is discarded. If a framing error occurs, the character is returned as an 8-bit null character (00H) without error indication.
6-8	Output parity control. For devices that support link parameters, the LINK$PARAMETER field (when enabled) overrides this parity setting.
	0 = Driver always sets parity bit to 0. This yields 8-bit data.
	1 = Driver always sets parity bit to 1. This yields 8-bit data.
	2 = Driver sets parity bit to give the byte even parity. This yields 7-bit data.
	3 = Driver sets parity bit to give the byte odd parity. This yields 7-bit data.
	4 = Driver does not alter the parity bit. This yields 8-bit data.
	5-7 Invalid.

NOTE

If bits 4-5 contain 2 or 3, and bits 6-8 also contain 2 or 3, then they must both contain the same value. That is, they must both reflect the same parity convention (even or odd).

Bits	Value and Meaning
9	OSC Translation control. 0 = Do not enable translation. 1 = Enable translation.
10	Terminal axes sequence control. This specifies the order in which Cartesian-like coordinates of elements on a terminal's screen are to be listed or entered. 0 = List or enter the horizontal coordinate first. 1 = List or enter the vertical coordinate first.
11	Horizontal axis orientation control. This specifies whether the coordinates on the terminal's horizontal axis increase or decrease as you move from left to right across the screen. 0 = Coordinates increase from left to right. 1 = Coordinates decrease from left to right.
12	Vertical axis orientation control. This specifies whether the coordinates on the terminal's vertical axis increase or decrease as you move from top to bottom across the screen. 0 = Coordinates increase from top to bottom. 1 =Coordinates decrease from top to bottom.
13-15	Reserved bits. For future compatibility, set to 0.

IN$RATE — WORD indicating the input baud rate. The word is encoded as follows:

0 = Invalid.

1 = Perform an automatic baud rate search.

	Other = Actual input baud rate, such as 9600.
OUT$RATE	WORD indicating the output baud rate. The word is encoded as follows: 0 or 1 = Use the input baud rate for output. Other = Actual output baud rate, such as 9600. Most applications require the input and output baud rates to be equal. In such cases, use IN$RATE to set the baud rate and specify a zero for OUT$RATE.
SCROLL$-NUMBER	WORD specifying the number of lines that are to be sent to the terminal each time the operator enters the appropriate control character for scrolling (Control-W is the default).

Depending on the requirements of your device, you can append additional driver-specific bytes to the TERMINAL$UNIT$INFORMATION structure.

6.2.4 Terminal Support Code (TSC) Data Area

DUIBs, Device Information Tables, and Unit Information Tables are structures that you or the ICU set up at configuration time to provide information about the initial state of your terminals. During configuration, the ICU assembles the DUIB into the Basic I/O System code segment and the Device Information and Unit Information Tables into the Terminal Support Code code segment. Therefore, they remain fixed throughout the life of the application system.

However, the Basic I/O System also provides a structure in the data segment (this section calls it the TSC Data Area) which changes to reflect the current state of the terminal controller and its units.

The TSC Data Area consists of three portions:

- A 30H-byte controller portion which contains information that applies to the device as a whole.
- A 400H-byte unit portion for each unit in the device. The NUM$UNITS field in the Device Information Table specifies the number of unit portions that the Basic I/O System creates.
- A user portion the user-written driver routines can use in any manner they choose (see the restriction below for MULTIBUS II message-passing terminal drivers). The DRIVER$DATA$SIZE field in the Device Information Table specifies the length of this portion. The USER$DATA$PTR field in the controller portion of the TSC data area points to the beginning of this field.

For message-passing drivers, the first two bytes of this field are structured as follows:

```
    DECLARE  DRIVER$DATA  STRUCTURE(
                            PORT$TOKEN        TOKEN,
                            OTHER$DATA(*)     BYTE);
```

where

PORT$TOKEN is the TOKEN for the port/mailbox used by the TSC to receive messages from the controller. The TSINITIO procedure creates this token; the TSFINISHIO procedure deletes it.

OTHER$DATA is a BYTE array available for driver-specific information.

Figure 6-2 illustrates the TSC Data Area graphically.

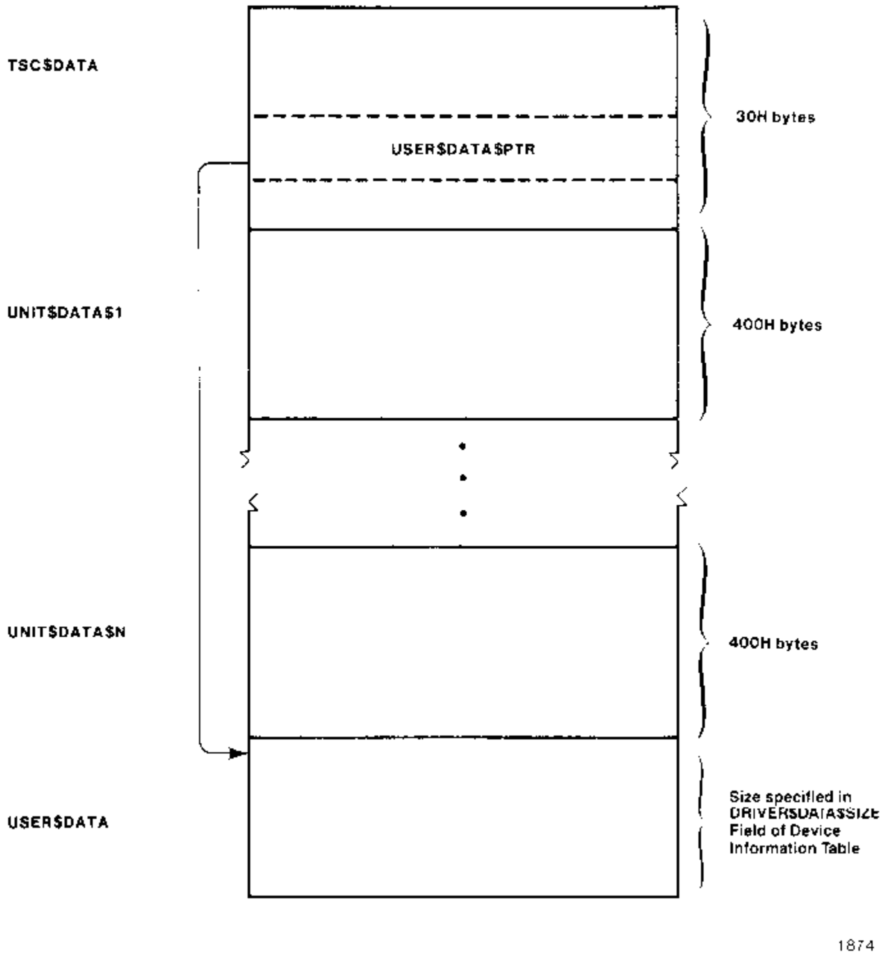


Figure 6-2. TSC Data Area

When the Basic I/O System calls one of your user-written driver procedures, it passes, as a parameter, a pointer either to the start of the TSC Data Area or to the start of one of the unit portions of the TSC Data Area. Your driver routines can then obtain information from the TSC Data Area or modify the information there.

A file named XTSDTN.LIT is located in /RMX286/IOS. This file declares the structure of the TSC Data Area, which always starts on a logical segment boundary, as follows:

```
DECLARE  TSC$DATA  STRUCTURE(
    IOS$DATA$SEGMENT              SELECTOR,
    STATUS                        WORD,
    INTERRUPT$TYPE                BYTE,
    INTERRUPTING$UNIT             BYTE,
    DEV$INFO$PTR                  POINTER,
    USER$DATA$PTR                 POINTER,
    RESERVED(34)                  BYTE);

DECLARE  UNIT$DATA(*)  STRUCTURE(
    UNIT$INFO$PTR                 POINTER,
    TERMINAL$FLAGS                WORD,
    IN$RATE                       WORD,
    OUT$RATE                      WORD,
    SCROLL$NUMBER                 WORD,
    X$Y$SIZE                      WORD,
    X$Y$OFFSET                    WORD,
    RAW$SIZE                      WORD,
    RAW$DATA$P                    POINTER,
    RAW$IN                        WORD,
    RAW$OUT                       WORD,
    OUTPUT$SCROLL$COUNT           WORD,
    UNIT$NUMBER                   BYTE,
    RESERVED(890)                 BYTE,
    BUFFERED$DEVICE$DATA(105)     BYTE);
```

where

IOS$DATA$-SEGMENT	A SELECTOR containing the base descriptor of the I/O System's data segment. The I/O System's terminal support routine TSINITIO fills in this information during initialization.
STATUS	A WORD in which the user-written terminal initialization procedure must return status information.
INTERRUPT$-TYPE	A BYTE in which the user-written terminal check procedure must return the encoded interrupt type. The possible values are

Bit	Meaning
0	None
1	Input interrupt
2	Output interrupt
3	Ring interrupt
4	Carrier interrupt

Bit	Meaning
5	Delay interrupt
6	Special character interrupt

If the terminal check procedure cannot guarantee there are no more interrupts to service, the terminal check procedure adds the following value to the encoded interrupt type it returns

8	More interrupts

For more information about these codes and their values, see the description of the terminal check procedure in the next section.

INTERRUPTING$-UNIT	A BYTE in which the user-written terminal check procedure must return the unit number of the interrupting device. This value identifies the unit that is interrupting.
DEV$INFO$PTR	A POINTER to the Terminal Device Information Table for this controller. The I/O System's terminal support routine TSINITIO fills in this data during initialization.
USER$DATA$PTR	A POINTER to the beginning of the user portion of the TSC Data Area. This user area can be used by the driver, as needed. The I/O System's terminal support routine TSINITIO fills in this pointer value during initialization.
RESERVED	Intel reserves this BYTE array for future use. Device drivers should not set these bytes.
UNIT$DATA	STRUCTUREs containing unit portions of the TSC Data Area. There is one structure for each unit (terminal) of the device. When a user attaches the unit (via the A$PHYSICAL$ATTACH$DEVICE system call or the ATTACHDEVICE Human Interface command, for example), the I/O System's terminal support routines initialize the appropriate UNIT$DATA structure. They perform the initialization by filling in all the fields of the UNIT$DATA structure with information from the DUIB and the Unit Information Table.
UNIT$INFO$PTR	A POINTER to the Unit Information Table for this terminal. This is the same information as in the UNIT$INFO$P field of the DUIB for this device-unit (terminal).

TERMINAL$-FLAGS- IN$RATE OUT$RATE SCROLL$NUMBER	The TSQUEUEIO procedure fills in these fields with information from the equivalent fields in the Unit Information Table when the unit is attached. Refer to the previous section, "Unit Information Table," for a description of these fields.
X$Y$SIZE	A WORD whose low-order byte specifies the number of character positions on each line of the terminal screen. The high-order byte specifies the number of lines on the terminal's screen. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls).
X$Y$OFFSET	A WORD whose low-order byte specifies the value that starts the numbering sequence of both the X and Y axis. The high-order byte specifies the value to which the numbering of the axes must fall back after reaching 127. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls). Refer to Chapter 2 for more information about cursor positioning.
RAW$SIZE	A WORD indicating the size of the unit's raw input buffer in bytes. The user-written terminal initialization procedure must set this size during initialization. Intel-supplied drivers for message-passing and nonbuffered devices always set this size to 256 decimal. Device drivers for interrupt-driven buffered devices set this value according to the size of the controller's on-board input buffer.
RAW$DATA$P	A POINTER to the unit's raw input buffer. The user-written terminal initialization procedure must initialize this pointer. For interrupt-driven buffered devices, this field should point to the controller's on-board input buffer. For message-passing and nonbuffered devices, this field should point to a logical segment the terminal initialization procedure creates.
RAW$IN	A WORD containing the offset from the RAW$DATA$P pointer indicating the head of the circular raw input buffer. The user-written terminal initialization procedure must initialize this value to zero. The terminal check procedure must update this value whenever characters are moved into the raw input buffer.

RAW$OUT	A WORD containing the offset from the RAW$DATA$P pointer indicating the tail of the circular raw input buffer. The user-written terminal initialization procedure must initialize this value to zero. The Terminal Support Code updates this value whenever it moves characters from the raw input buffer to the type-ahead buffer. The device driver should use the difference between RAW$IN and RAW$OUT to determine how many characters are in the raw input buffer. After initialization, the driver must never update RAW$OUT.
OUTPUT$-SCROLL$COUNT	A WORD the Terminal Support Code updates to indicate the number of output lines that have been displayed while in scrolling mode. The terminal driver should not update this count. Nonbuffered terminal drivers should not change this value. Buffered terminal drivers must decrement this number, in the TERM$UTILITY function 0, by the number of lines actually output.
UNIT$NUMBER	A BYTE the Terminal Support Code fills in with the unit number of this unit.
RESERVED	Intel reserves this BYTE array for future use. Device drivers should not set these bytes.
BUFFERED$DE-VICE$DATA	BYTEs containing additional information that applies to drivers of buffered devices (intelligent communications processors that maintain their own internal memory buffers). Refer to the "Additional Information for Buffered Devices" section to see how to access these bytes.

6.2.5 Additional Information for Buffered Devices

A buffered communications device is an intelligent communications processor, such as the iSBC 544A and iSBC 186/410 boards, that manages its own buffers of data separately from the ones managed by the Terminal Support Code. An off-board message-passing terminal controller must be a buffered device.

Interrupt-driven buffered device drivers differ from message-passing buffered device drivers in how they manage the raw input buffer. Because MULTIBUS I supports a shared-memory architecture, an interrupt-driven terminal driver can use the dual-port input buffer on the controller as the raw input buffer. However, since MULTIBUS II supports connectionless data transfers, a message-passing terminal driver must maintain its own circular raw input buffer in addition to the controller's input buffer. A MULTIBUS II controller uses the MULTIBUS II Transport Protocol to send data (via messages) to the terminal driver, which transfers the data to the raw input buffer it maintains. Likewise, an interruptless MULTIBUS I message-passing terminal driver uses a mailbox to send or receive data from another job that manages data input and output. Subsequently, the Terminal Support Code transfers the data from the driver's raw input buffer to its type-ahead buffer. (For more information on message-passing, see the description of the Nucleus Communications Service in the *Extended iRMX II Nucleus User's Guide.*)

If you are writing a driver for a buffered communications device, your driver routines must make use of the BUFFERED$DEVICE$DATA fields of the UNIT$DATA structure. In so doing, they should impose the following structure on those 105 bytes:

```
DECLARE   BUFFERED$DEVICE$DATA  STRUCTURE(
                                 BUFFERED$DEVICE          BYTE,
                                 BUFF$INPUT$STATE         WORD,
                                 BUFF$OUTPUT$STATE        WORD,
                                 SELECT(2)                BYTE,
                                 LINE$RAM$P               POINTER,
                                 FUNCTION$ID              BYTE,
                                 IN$COUNT                 BYTE,
                                 OUT$COUNT                WORD,
                                 UNITS$AVAILABLE          WORD,
                                 OUTPUT$BUFFER$SIZE       WORD,
                                 USER$BUFFER$P            POINTER,
                                 ECHO$COUNT               BYTE,
                                 ECHO$BUFFER$P            POINTER,
                                 RECEIVED$SPECIAL         WORD,
                                 SPECIAL$MODES            WORD,
                                 HIGH$WATER$MARK          WORD,
                                 LOW$WATER$MARK           WORD,
                                 FC$ON$CHAR               WORD,
                                 FC$OFF$CHAR              WORD,
                                 LINK$PARAMETER           WORD,
                                 SPC$HI$WATER$MARK        WORD,
                                 SPECIAL$CHAR(4)          BYTE,
                                 RESERVED(25)             BYTE,
                                 DRIVER$USE$ONLY(32)      BYTE);
```

where

BUFFERED$-DEVICE	A BYTE which the driver's terminal setup procedure sets to TRUE (0FFH) to indicate that the unit is a buffered device. If BUFFERED$DEVICE is set to FALSE (00H), none of the rest of the fields in this structure are meaningful.
BUFF$INPUT$-STATE	A WORD used for passing information about the input state between the Terminal Support Code and the terminal driver. The bits of this word are encoded as follows (bit 0 is the low-order bit):

Bits	Value and Meaning
0	The Terminal Support Code sets this bit to indicate whether a modem is online. When set to one, a modem is online and the driver should set DTR. When set to zero, no modem is online and the driver should reset DTR. The Terminal Support Code calls the terminal utility procedure to set or reset DTR.
1	The Terminal Support Code sets this bit after taking characters from the raw input buffer. It then calls the terminal utility procedure, which should reset the bit after informing the firmware about the removal of the characters. For example, the iSBC 544A driver sends an input command to the firmware and, when an interrupt indicates that the command has completed, resets this bit.
2, 4, 6,7	Reserved. The driver should not set these bits.
3	This bit should be cleared by the driver whenever it sends an input command to the firmware; otherwise, the TSC will not accept characters from the raw buffer, if a type-ahead-buffer-full condition previously existed. The TSC will set this flag when it finds the type-ahead buffer full; when it is no longer full, the TSC will call for an input command from the driver. The driver must clear the bit at this time.

Bits	Value and Meaning
5	The Terminal Support Code sets this bit, based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls), to indicate whether output control characters are being processed. If set to zero, the Terminal Support Code ignores output control characters as they appear in the input stream. If set to one, the Terminal Support Code processes output control characters in the input stream. The device driver can examine this bit and, if the controller's firmware has the capability, direct the firmware to process output control characters when they appear in the input stream.
8-15	Available bits. The driver can use these bits to keep track of its input state. The Terminal Support Code does not check or set these bits.

BUFF$OUTPUT$-STATE A WORD used for passing information about the output state between the Terminal Support Code and the terminal driver. The bits of this word are encoded as follows (bit 0 is the low-order bit):

Bits	Value and Meaning
0	The terminal initialization or terminal setup procedure sets this bit to indicate whether the Terminal Support Code or the output device keeps track of the number of characters available in the device's output buffer. If set to zero, the Terminal Support Code maintains this count without requiring information from the device. If set to one, the terminal driver (or the device's firmware) must keep track of the space remaining in the output buffer. If the device is maintaining this information, the terminal utility procedure must place into the UNITS$AVAILABLE field of this structure the number of bytes of free space remaining in the output buffer.
1	The Terminal Support Code sets this bit, based on output control characters entered by the operator, to indicate the output state. If set to zero, output can occur. If set to one, output is stopped. The device driver must examine this bit when sending output.

	2	The Terminal Support Code sets this bit, based on output control characters entered by the operator, to indicate whether the output device is in scrolling mode. If set to one, the device is in scrolling mode (only a certain number of characters appear on the screen; the operator must press a key to see the next group of characters). If set to zero, scrolling mode is not in effect (characters appear on the screen without stopping). The device driver must examine this bit when sending output.
	3-7	Reserved. Device drivers should not set these bits.
	8-15	Available to the device driver for keeping track of its output state. The Terminal Support Code does not set or check these bits.

SELECT(2)
A BYTE array the driver's terminal initialization procedure must fill in to identify the board and line number of this unit. The first byte identifies the number of this unit's controller board (where 0 is the first board). The second byte identifies the line number on that board (where the first line is line 0).

LINE$RAM$P
For interrupt-driven devices, a POINTER to the dual-port RAM address of the specified line. The driver's terminal initialization procedure must place this address here so that it doesn't need to calculate the address each time it accesses the unit.

For message-passing devices, this parameter is ignored.

FUNCTION$ID
A BYTE the Terminal Support Code fills in specifying a function the driver's terminal utility procedure should perform. Refer to the description of the terminal utility procedure for a description of the functions the Terminal Support Code can request.

IN$COUNT
A BYTE the Terminal Support Code fills in when calling the driver's terminal utility procedure with the FUNCTION$ID field set to 1. This field specifies the number of bytes the Terminal Support code has moved from the raw input buffer to the Terminal Support Code's buffer.

OUT$COUNT
A WORD the Terminal Support Code fills in when calling the driver's terminal utility procedure with the FUNCTION$ID field set to 0. This field specifies the number of bytes to be moved from the user's output buffer to the device's on-board output buffer. This field must be decremented by the number of bytes actually output.

UNITS$-AVAILABLE	A WORD used by drivers keeping track of the number of characters remaining in their output buffers. When the Terminal Support Code calls the terminal utility procedure to request the number of characters remaining (FUNCTION$ID set to 5), the terminal utility procedure must place that number in this field.
OUTPUT$-BUFFER$SIZE	A WORD listing the size of the buffered unit's output buffer. The driver's terminal initialization procedure must set this word to the correct value.
USER$BUFFER$P	A POINTER to the user's buffer. When the Terminal Support Code calls the driver's terminal utility procedure to transmit output characters (FUNCTION$ID set to 0), that procedure must transfer the number of characters specified in OUT$COUNT from this user buffer to the unit's output buffer.
ECHO$COUNT	A BYTE in which the Terminal Support Code places the number of characters the device's terminal utility procedure should echo to the terminal (when FUNCTION$ID is set to 8). The terminal utility procedure gets these characters from the ECHO$BUFFER$P buffer.
ECHO$BUFFER$P	A POINTER in which the Terminal Support Code places the address of the characters to be echoed to the terminal (when FUNCTION$ID is set to 8). The device's terminal utility procedure echoes these characters to the terminal.
RECEIVED$-SPECIAL	This WORD is used by devices supporting Special Character mode. When Special Character Mode is enabled and a special character interrupt occurs, the driver's terminal check procedure sets this field to indicate which special character was entered. The terminal check procedure sets the low-order four bits of this word (bits 0 through 3) to indicate which special character was entered. Bit 0 corresponds to the first character defined in the SPECIAL$CHAR array. Bit 1 corresponds to the second character, and so forth. The driver can ignore the other 12 bits.
SPECIAL$MODES	A WORD indicating whether the terminal is using any special modes. The Terminal Support Code sets this word based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls). Encode the bits of this word as follows (bit 0 is the low-order bit):

Bits	Value and Meaning
0	Flow Control Mode. This bit specifies whether the communications board sends flow control characters (selected by FC$ON$CHAR and FC$OFF$CHAR, but usually XON and XOFF) to turn input on and off. The settings are as follows: 0 = Disable flow control. 1 = Enable flow control. When flow control is enabled, the communication board can control the amount of data sent to it to prevent buffer overflow.
1	Special Character Mode. This bit is used in conjunction with the SPC$HI$WATER$MARK field to specify whether the Terminal Support Code responds to special characters immediately as they are typed (Special Character Mode) or whether the characters are handled when received through the normal input stream. If your device supports special characters (currently, only the iSBC 188/48, iSBC 188/56, iSBC 186/410, iSBC 546, iSBC 547, and iSBC 548 boards do), the device can send an interrupt whenever a special character (defined later in the SPECIAL$CHAR array) is typed. When Special Character Mode is on, the device's terminal check procedure sets the RECEIVED$SPECIAL field whenever a special character interrupt occurs. If the special character is defined as a signal character, the Terminal Support Code sends a unit to the appropriate signal semaphore. The setting of this bit is as follows: 0 = Disable Special Character Mode. 1 = Enable Special Character Mode. The SPC$HI$WATER$MARK field is used in conjunction with this bit to control Special Character Mode.

	2-15	Reserved bits. The device driver should not set these bits.

HIGH$WATER$-MARK	When the communication board's input buffer fills to contain the number of bytes specified in this WORD, the board sends the flow control character to stop input. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls).
LOW$WATER$-MARK	When the number of bytes in the communication board's input buffer drops to the number specified in this WORD, the board sends the flow control character to start input. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls).
FC$ON$CHAR	A WORD specifying an ASCII character the communication board sends to the connecting device when the number of bytes in its buffer drops to the value in LOW$WATER$MARK. Normally this character tells the connecting device to resume sending data. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls).
FC$OFF$CHAR	A WORD specifying an ASCII character the communication board sends to the connecting device when the number of characters in its buffer rises to the value in HIGH$WATER$MARK. Normally this character tells the connecting device to stop sending data. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls).
LINK$-PARAMETER	A WORD specifying the characteristics of the physical link between the terminal and a device. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls). Physical link parameters are not supported by all devices or device drivers. For supported drivers (such as the Terminal Communications Controller driver), when the physical link parameters are used (bit 15 set to one), the Terminal Support Code passes the low-order byte of the LINK$PARAMETER field to the driver, which passes it directly to the controller. The controller sets the physical link appropriately.

When the physical link parameters are used, they override the setting of input and output parity in the TERMINAL$FLAGS field. When the physical link parameters are not used, the TERMINAL$FLAGS fields apply.

The bits of this word are encoded as follows (bit 0 is the low-order bit):

Bits	Value and Meaning
0-1	The value in these bits specifies the input and output parity, as follows: 0 = No parity 1 = Invalid value 2 = Even parity 3 = Odd parity
2-3	The value in these bits specifies the character length, as follows: 0 = 6 bits/character 1 = 7 bits/character 2 = 8 bits/character 3 = 5 bits/character
4-5	The value in these bits indicates the number of stop bits, as follows: 0 = 1 stop bit 1 = 1-1/2 stop bits 2 = 2 stop bits 3 = Reserved--drivers should not set this bit
6	The value in this bit specifies how the transmitter and receiver are enabled, as follows: 0 = Transmitter and receiver unconditionally enabled 1 = CTS is transmitter-enable; CD is receiver-enable
7-8	The value in these bits specifies how receive errors (parity, framing, or overrun errors) are handled, as follows:

Bits	Value and Meaning
	0 = Replace erroneous character by ASCII NUL (0H).
	1 = Discard erroneous character.
	2 = Prefix erroneous character by the two-byte sequence 0FFH 00H. A valid 0FFH character will be replaced by the two-character sequence 0FFH 0FFH.
	3 = Set the most significant bit of erroneous character to 1.
9-14	Reserved. Drivers should not set these bits.
15	Determines whether or not the link parameters are used, as follows:
	0 = The link parameters are not used. The Terminal Support Code does not pass the low-order byte of the LINK$PARAMETER field to the controller. The input and output parity applies from the setting of TERMINAL$FLAGS.
	1 = The link parameters are used. The Terminal Support Code passes the low-order byte of the LINK$PARAMETER field to the controller, overriding the parity settings in TERMINAL$FLAGS.

SPC$HI$WATER$-MARK

A WORD used in conjunction with the SPECIAL$MODES field to control Special Character Mode. When the device's input buffer fills to contain the number of characters specified in this field, Special Character Mode is enabled (assuming the SPECIAL$MODES field has Special Character Mode turned on). If the number of characters in the device's input buffer is less than the high water mark, Special Character Mode is disabled, even if it is turned on in the SPECIAL$MODES field. The Terminal Support Code sets this field based on user input (OSC sequences, A$SPECIAL calls, or S$SPECIAL calls).

If the Special Character Mode is turned off in the SPECIAL$MODES field, this field has no effect.

SPECIAL$-CHAR(4)	A BYTE array containing as many as four characters that are defined as the device's special characters. If Special Character Mode is on, typing any of these characters at the keyboard generates a special-character interrupt. When this happens, the driver's terminal check procedure sets the RECEIVED$SPECIAL field of this structure to indicate which special character was typed. If the character is a signal character, the Terminal Support Code processes it immediately. The Terminal Support Code sets this field based on user input (OSC sequences). If you define less than four special characters, you must fill the remaining slots in the array with duplicates of the last character you define.
RESERVED(25)	Intel reserves this BYTE array for future use. Device drivers should not set these bytes.
DRIVER$USE$-ONLY(32)	A BYTE array reserved for use by the device driver. The Terminal Support Code does not read or write these bytes.

6.3 TSC PROCEDURES TERMINAL DRIVERS MUST CALL

Some user-written message-passing terminal drivers make calls to TSC routines. The following paragraphs describe the routines briefly. Sections that follow describe the routines in more detail.

A terminal mutual exclusion procedure. The user-supplied terminal check procedure calls this TSC procedure to provide mutual exclusion for the unit data structure of the message-sending device.

A terminal set output buffer size procedure. The user-supplied terminal initialization procedure calls this TSC procedure to communicate the size of the controller's output buffer to the Terminal Support Code.

6.3.1 Terminal Mutual Exclusion Procedure

Used by message-passing drivers, this TSC-provided procedure gains exclusive access to the UNIT$DATA structure for the message-sending device. This procedure must be declared as an external procedure with one pointer parameter and be called by the user-supplied terminal check procedure.

The syntax of a call to the TSC terminal mutual exclusion procedure is as follows:

```
CALL ts$mutex$unit(unit$data$p);
```

where

unit$data$p	POINTER to the UNIT$DATA structure for the message-sending unit. The terminal check procedure obtains this value by using the pointer to the TSC Data Area.

6.3.2 Terminal Set Output Buffer Size Procedure

If the terminal initialization procedure does not inform the TSC of the controller's output buffer size, the user-supplied driver must call this procedure to do so when this information becomes available to it. For example, a message-passing driver that can determine the size of the controller's output buffer only after the unit is attached must call this procedure.

The user-supplied driver must declare the TS$SET$OUT$BUF$SIZE procedure as an external procedure with one pointer parameter. The syntax of a call to the TSC terminal set output buffer size procedure is as follows:

```
CALL ts$set$out$buf$size(udata$p, out$buf$size);
```

where

udata$p	POINTER to the UNIT$DATA structure for the attached unit.
out$buf$size	WORD containing the controller's output buffer size for this unit.

6.4 PROCEDURES TERMINAL DRIVERS MUST SUPPLY

The routines that make up the Basic I/O System's Terminal Support Code constitute the bulk of the terminal device driver. These routines, in turn, make calls to device-dependent routines that you must supply. The following paragraphs describe the routines briefly. Sections that follow describe the routines in more detail.

A terminal initialization procedure. This procedure must perform any initialization functions necessary to get the terminal controller ready to process I/O requests. TSINITIO calls this procedure.

A terminal finish procedure. This procedure must perform any final processing so that the terminal controller can be detached. TSFINISHIO calls this procedure.

A terminal setup procedure. This procedure sets up the terminal in the proper mode (baud rate, parity, etc.). TSQUEUEIO and the Terminal Support Code's interrupt/message task calls this procedure.

A terminal answer procedure. This procedure sets the Data Terminal Ready (DTR) line for modem support. TSQUEUEIO and the Terminal Support Code's interrupt/message task calls this procedure.

A terminal hangup procedure. This procedure clears the Data Terminal Ready (DTR) line for modem support. TSQUEUEIO and the Terminal Support Code's interrupt/message task calls this procedure.

A terminal check procedure. This procedure determines which terminal sent an interrupt signal and what type of interrupt it is. The Terminal Support Code's interrupt handler/message task calls this procedure.

NOTE

In interrupt-driven systems, the length of the terminal check procedure affects interrupt latency since it is called from an interrupt handler and runs with interrupts disabled. In message-passing systems, the length of this procedure does not affect interrupt latency since it is called from a task and runs with interrupts enabled.

A terminal output procedure. This procedure displays a character at a terminal. TSQUEUEIO and the Terminal Support Code's interrupt task call this procedure.

A terminal utility procedure. The Terminal Support Code calls this procedure to perform buffered-device operations.

When the Terminal Support Code calls these procedures, it passes, as a parameter, a pointer to the TSC Data Area described in the previous section. If the called procedure is to perform duties on behalf of all the terminals connected to the controller, the Terminal Support Code passes a pointer to the beginning of the TSC Data Area (the device portion). However, if the procedure is to perform duties for just a particular terminal, the Terminal Support Code passes a pointer to the unit portion of the TSC Data Area that corresponds to the terminal.

Because the TSC Data Area starts at the beginning of a logical segment, a procedure that receives a pointer to a unit portion of the data area can construct a pointer to the beginning of the TSC Data Area. It does this by calling the PL/M-286 built-in procedure BUILD$PTR using the base part of the pointer it received and an offset of 0. Also, if a procedure, such as the terminal check procedure, receives a pointer to the beginning of the TSC data area, it can calculate where any unit portion of the data area starts by using the following formula:

```
unit$data$p = base(of TSC data area):[30H + (unit number * 400H)]
```

6.4.1 Terminal Initialization Procedure

This procedure must initialize the controller. The nature of this initialization is device-dependent. When finished, the terminal initialization procedure must fill in the STATUS field of the TSC Data Area, as follows:

- If initialization is successful, it must set STATUS to E$OK (0).
- If initialization is not successful, it should normally set STATUS equal to E$IO (2BH). However, it can set the STATUS field to any other value, in which case the Basic I/O System returns that value to the task that is attempting to attach the device. (The Human Interface ATTACHDEVICE command expects the procedure to return the E$IO status if initialization is unsuccessful.)

In addition, the terminal initialization procedure must initialize the raw input buffer for each unit of the device. How this operation is performed depends on whether your system is interrupt-driven or message-based and whether the device is buffered or nonbuffered.

- **Interrupt-driven buffered devices.** For each configured unit, the terminal initialization procedure must place a pointer to the unit's on-board input buffer in the RAW$DATA$P field of that unit's UNIT$DATA portion of the TSC Data Area.

  It must also set the RAW$SIZE field to the size of the input buffer, and it should initialize RAW$IN and RAW$OUT to zero for the start of the input buffer.

  Finally, it must set the OUTPUT$BUFFER$SIZE field of the buffered device's BUFFER$DEVICE$DATA structure to the size of the unit's output buffer, and the BUFFERED$DEVICE field to TRUE to inform the Terminal Support Code to use this buffer size. To preserve the ATTACH and DETACH signaling protocol, Terminal Support Code will set the BUFFERED$DEVICE field to FALSE.
- **Message-passing and nonbuffered devices.** For each configured unit, the terminal initialization procedure must create a logical segment for the unit's raw input buffer. Then it must place a pointer to the appropriate segment in the RAW$DATA$P field of that unit's UNIT$DATA portion of the TSC Data Area. It must also place the size of the segment in the RAW$SIZE field, and it must initialize the RAW$IN and RAW$OUT fields to zero (the offset for the start of the segment).

  Although the terminal initialization procedure can choose any size for the raw input buffer, it should not choose a value that is too large. (For nonbuffered devices, when the type-ahead buffer fills up, the Terminal Support Code discards any bytes in the raw input buffer that do not fit in the type-ahead buffer, so keeping the raw input buffer smaller minimizes data loss from type-ahead.) The recommended size for message-passing devices and nonbuffered devices is 256 decimal bytes. The raw input buffer should never be larger than 256 decimal bytes, because that is the size of the Terminal Support Code's type-ahead buffer.

For message-passing drivers, the terminal initialization procedure must create the port/mailbox the Terminal Support Code uses to receive messages. This token is passed to the Terminal Support Code by PORT$TOKEN in the driver data portion of the TSC Data Area.

The syntax of a call to the user-written terminal initialization procedure is as follows:

```
CALL term$init(tsc$data$ptr);
```

where

term$init	Name of the terminal initialization procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
tsc$data$ptr	POINTER to the beginning of the TSC Data Area.

6.4.2 Terminal Finish Procedure

The Terminal Support Code calls this procedure when a user detaches the last terminal unit on the terminal controller. The terminal finish procedure can simply do a RETURN; it can clean up data structures for the driver, or it can clear the controller. It should delete any objects created by the other terminal procedures. The syntax of a call to the user-written terminal finish procedure is as follows:

```
CALL term$finish(tsc$data$ptr);
```

where

term$finish	Name of the terminal finish procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
tsc$data$ptr	POINTER to the beginning of the TSC Data Area.

6.4.3 Terminal Setup Procedure

This procedure "sets up" a terminal according to the fields in the corresponding UNIT$DATA portion of the TSC Data Area. The Terminal Support Code calls this procedure when attaching the unit the first time, when detaching the device (for buffered devices only), and whenever the terminal's input baud rate, output baud rate, read parity checking, and write parity checking attributes are changed and the line must be reinitialized.

When the terminal setup procedure receives control, it should initialize the unit using the information that already exists in the UNIT$DATA portion of the TSC Data Area.

If IN$RATE is 1, then the terminal setup procedure must start a baud rate search. (The terminal check procedure usually finishes the search and then fills in IN$RATE with the actual baud rate.) If OUT$RATE is 0 or 1, the terminal setup procedure assumes the output baud rate is the same value as the input baud rate.

If the terminal controller is a buffered device, the terminal setup procedure must set the BUFFERED$DEVICE field to TRUE (0FFH). It should also fill in the other fields of the BUFFERED$DEVICE$DATA structure (refer to the "Additional Information for Buffered Devices" section). In addition, it should enable the communication device's on-board receiver interrupt (the one for the unit being attached) so that it can accept data from the connected terminal.

When a user detaches a unit on a buffered device, the Terminal Support Code sets the BUFFERED$DEVICE field to FALSE (0H) and again calls the terminal setup procedure. The terminal setup procedure should disable the communication device's on-board receiver interrupt (the one for the unit being detached) to prevent extraneous characters from being received.

To distinguish between an "attach device", a "detach device", and a "change terminal characteristics" operation requiring reinitialization, the terminal setup procedure should establish its own internal flag (one for each unit) in addition to the BUFFERED$DEVICE fields. A user bit in BUFF$OUTPUT$STATE can be used for this flag. The terminal setup procedure can use its internal flag as follows:

1. Initially, the terminal initialization procedure sets the flag of each unit to FALSE to indicate that no devices are attached.
2. When the Terminal Support Code calls the terminal setup procedure to attach a unit, both the BUFFERED$DEVICE field and the internal flag are FALSE. The terminal setup procedure recognizes from this combination that the operation is an "attach device."
3. The terminal setup procedure performs the "attach device" operations and sets the internal flag and the BUFFERED$DEVICE flag to TRUE to indicate that the device is attached.
4. When the Terminal Support Code calls the terminal setup procedure after attaching the unit but before detaching it, both the BUFFERED$DEVICE field and the internal flag are TRUE. This combination means that the line parameters (such as baud rate or parity) have changed. The terminal setup procedure must reinitialize the unit with the correct characteristics.
5. When the unit is detached, the Terminal Support Code sets the BUFFERED$DEVICE flag to FALSE and calls the terminal setup procedure. In this situation, the BUFFERED$DEVICE field is FALSE, but the internal flag is TRUE. The terminal setup procedure recognizes from this combination that the operation is a "detach device."

If your terminal driver supports a modem, the terminal setup procedure should also set the Data Terminal Ready line to active. Refer to the "Terminal Hangup" section for more information.

When a unit of a nonbuffered device is initialized, the terminal setup procedure should notify the Terminal Support Code that the unit is ready to accept interrupts. It does this by calling the following procedure provided by the Terminal Support Code:

```
CALL xts$set$output$waiting(unit$data$p);
```

where unit$data$p is the pointer to this unit's UNIT$DATA portion of the TSC Data Area. The terminal setup procedure must declare the XTS$SET$OUTPUT$WAITING procedure as an external procedure with one pointer parameter. For buffered devices, the XTS$SET$OUTPUT$WAITING procedure does not need to be called.

The syntax of a call to the user-written terminal setup procedure is as follows:

```
CALL term$setup(unit$data$n$ptr);
```

where

term$setup	Name of the terminal setup procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$ptr	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area.

6.4.4 Terminal Answer Procedure

This procedure activates the Data Terminal Ready (DTR) line for a particular terminal. The Terminal Support Code calls the terminal answer procedure only when both of the following conditions are true:

- Bit 3 of TERMINAL$FLAGS in the terminal's UNIT$DATA structure (the modem indicator) is set to 1.
- The Terminal Support Code has received a Ring Indicate signal (the phone is ringing) or an answer request (via an OSC modem answer sequence) for the terminal. Refer to Chapter 2 for more information about OSC sequences.

The syntax of a call to the user-written terminal answer procedure is as follows:

```
CALL term$answer(unit$data$n$p);
```

where

term$answer	Name of the terminal answer procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$p	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area.

6.4.5 Terminal Hangup Procedure

This procedure clears the Data Terminal Ready (DTR) line for a particular terminal. The Terminal Support Code calls the terminal hangup procedure only when both of the following are true:

- Bit 3 of TERMINAL$FLAGS in the terminal's UNIT$DATA structure (the modem indicator) is set to 1.
- The Terminal Support Code has received a Carrier Loss signal (the phone is hung up) or a hangup request (via an OSC modem hangup sequence) for the terminal. Refer to Chapter 2 for more information about OSC sequences.

The syntax of a call to the user-written terminal hangup procedure is as follows:

```
CALL term$hangup(unit$data$n$p);
```

where

term$hangup	Name of the terminal hangup procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$p	POINTER to the terminal's UNIT$DATA structure in the Terminal Support Code data Area.

NOTE

Some modem devices recognize only carrier detect as an indication that someone is calling and loss of carrier detect as an indication of hangup. However, most of these devices require the Data Terminal Ready line to be active before they can recognize carrier detect. For these devices, the terminal setup procedure must activate the Data Terminal Ready line. Likewise, the terminal hangup procedure must clear the Data Terminal Ready line for about one second then reactivate it.

6.4.6 Terminal Check Procedure

For interrupt-driven systems, the Terminal Support Code calls this procedure whenever the device generates an interrupt, which usually indicates that a key on that terminal's keyboard has been pressed. When called, the terminal check procedure should determine the kind of interrupt and the interrupting unit, as follows:

1. Check all terminals on the device for an input character. If found, put the input character in the unit's raw input buffer, updating RAW$IN accordingly.

2. If no input character is available, check to see if any device is ready to transmit another character to the terminal.
3. If no device is ready to transmit a character to the terminal, and if this is a buffered device for which special character mode is enabled, check for a special character.
4. If no special character is available, check for a change in status (such as a ring or carrier interrupt).

When the terminal check procedure finds the first valid interrupt, it should quit scanning other units. Then it should place the unit number of the interrupting unit in the INTERRUPTING$UNIT field of the TSC Data Area.

For message-passing drivers, the Terminal Support Code receives a message from the controller, then it calls the terminal check procedure to identify the unit sending the message and to process the message, as follows:

1. Examine the received message and place the unit number of the sending unit in the INTERRUPTING$UNIT field of the TSC Data Area.
2. To provide mutual exclusion to this unit's data structure, call the TSC's terminal mutual exclusion procedure. For a description of this procedure see the "TSC Procedures Terminal Drivers Must Call" section in this chapter.
3. Copy any received characters into the device driver's raw input buffer (remember this is a circular buffer). Modify the parity bits appropriately. Update the head pointer of the raw input buffer.
4. Perform any immediate processing appropriate to the received message. This step may involve sending commands to the controller.
5. Place the INTERRUPT$TYPE corresponding to the message in the TSC Data Area.

For interrupt-driven and message-passing systems, place the type of interrupt this procedure will return in the INTERRUPT$TYPE field of the TSC Data Area. For both types of systems, the Terminal Support Code expects the following values in this field:

0	No interrupt occurred.
1	An input interrupt occurred.
2	An output interrupt occurred. This signals the Terminal Support Code to call the terminal output procedure to display the output character at the terminal.
3	A ring interrupt occurred. If the TERMINAL$FLAGS field in the unit's UNIT$DATA structure indicates that the unit supports a modem, this signals the Terminal Support Code to call the terminal answer procedure to activate the Data Terminal Ready (DTR) line.
4	A carrier-loss interrupt occurred. If the TERMINAL$FLAGS field in the unit's UNIT$DATA structure indicates that the unit supports a modem, this signals the

Terminal Support Code to call the terminal hangup procedure to reset the DTR line.

5	A baud rate scan is in progress and the terminal setup procedure needs more time to determine the baud rate. This signals the Terminal Support Code to delay for some time and call the terminal setup procedure again.
6	A special-character interrupt occurred. Only certain controllers, such as the iSBC 188/48 intelligent communications controller, can generate these interrupts. The terminal check procedure sets the RECEIVED$SPECIAL field of the device's BUFFERED$DEVICE$DATA structure to identify the character. If the character is a signal character, this interrupt type directs the Terminal Support Code to send a unit to the appropriate signal semaphore.

If the device uses only a single interrupt, additional interrupt-causing events can occur while the original interrupt is being processed. To avoid missing these occurrences, the terminal check procedure must add the following value to the value it places in the INTERRUPT$TYPE field:

8	More interrupts are available.

Adding this value signals the Terminal Support Code to call the terminal check procedure again after it processes the current interrupt.

Unless the controller hardware guarantees that an additional interrupt will be set after one of multiple pending interrupts is serviced, the terminal check procedure should always signal that more interrupts are available unless it cannot detect interrupts at all. That is, it should always return one of the following values in the INTERRUPT$TYPE field:

0H	No interrupt occurred.
9H	An input interrupt occurred, and more interrupts are available.
0AH	An output interrupt occurred, and more interrupts are available.
0BH	A ring interrupt occurred, and more interrupts are available.
0CH	A carrier-loss interrupt occurred, and more interrupts are available.
0DH	The terminal check procedure wasn't able to determine the baud rate yet. Call terminal setup again. More interrupts are available.
0EH	A special character interrupt occurred, and more interrupts are available.

By returning these values, the terminal check procedure ensures the Terminal Support Code calls it again. Otherwise, the driver could lose characters. If there are no more interrupts to service, the terminal check procedure can return a zero value (no interrupt) the last time it is called.

If your terminal driver supports a baud rate search to determine the baud rate of an individual terminal, the terminal check procedure must ascertain the terminal's baud rate, as follows:

1. The first time the terminal check procedure encounters an input interrupt for a particular terminal, it should examine the IN$RATE field of that terminal's UNIT$DATA structure to determine the baud rate.
2. If the IN$RATE field is set to 1 (perform automatic baud rate search), the terminal check procedure should examine the input character to determine if it is an uppercase "U". (It can usually check for 19200, 9600, and 4800 baud in one attempt.)
3. If the terminal check procedure determines the baud rate, it should set the IN$RATE field of the UNIT$DATA structure to reflect the actual input baud rate and skip Steps 4 and 5.
4. If the terminal check procedure cannot determine the baud rate, it should increment the IN$RATE field in the UNIT$DATA structure. When the next input interrupt occurs, the terminal check procedure can try again to determine the baud rate. Refer to the example terminal driver on the "Examples" diskette to see how to implement a baud rate scan.
5. Place a value of 0DH in the INTERRUPT$TYPE field (delay interrupt plus more). The 0DH value tells the Terminal Support Code that a baud rate scan is in progress. The Terminal Support Code then waits a few clock cycles and calls the terminal setup procedure to "set up" the terminal for the new baud rate. When the next interrupt occurs, the terminal check procedure can continue with the baud rate scan.

If the terminal check procedure encounters an input interrupt, its additional action depends on whether it is supporting a buffered or nonbuffered device.

For message-passing devices and nonbuffered devices, the terminal check procedure must also read the input character, adjusting the parity bit according to bits 4 and 5 of the TERMINAL$FLAGS field in the interrupting unit's UNIT$DATA structure, and move that input character into the raw input buffer pointed to by the RAW$DATA$P field of the UNIT$DATA structure. When RAW$IN equals RAW$OUT minus 1, the circular buffer is full. Message-passing devices can handle up to 256 characters per message. Nonbuffered devices handle one character per interrupt.

For interrupt-driven buffered devices, the terminal check procedure does not read the input character(s). Rather, the Terminal Support Code will call the terminal utility procedure (with a function ID of 1) to retrieve characters from the buffered device. If the device is capable of informing the Terminal Support Code about the current values of RAW$IN and RAW$OUT, the terminal check procedure doesn't need to keep track of RAW$IN. Later the Terminal Support Code will call the terminal utility procedure (with a function ID of 0AH) to update the RAW$IN field.

However, if the interrupt-driven buffered device is not capable of informing the driver about the current values of the RAW$IN and RAW$OUT fields, the terminal check procedure must keep track of the RAW$IN value. It can either update the RAW$IN field in the UNIT$DATA structure each time an input interrupt occurs (in which case function ID 0AH of the terminal utility procedure causes a null operation), or it can maintain an internal copy of RAW$IN and make the information available to the terminal utility procedure.

If the interrupt is a special character interrupt, the terminal check procedure must set the SPECIAL$RECEIVED field of the UNIT$DATA structure to identify the special character. The four possible special characters are listed in the SPECIAL$CHAR array of the UNIT$DATA structure. If the special character received is the first character listed in the SPECIAL$CHAR array, the terminal check procedure should set bit zero of the SPECIAL$RECEIVED field. If the special character is the second character listed in SPECIAL$CHAR, the terminal check procedure should set bit 1 of SPECIAL$RECEIVED. Bits 2 and 3 of SPECIAL$RECEIVED correspond to the third and fourth special characters listed.

For interrupt-driven drivers, the syntax of the call to the user-written terminal check procedure is as follows:

```
CALL term$check(tsc$data$ptr);
```

For message-passing drivers, the syntax of the call to the terminal check procedure is as follows:

```
CALL term$check(tsc$data$ptr,message$ptr);
```

where

term$check	The name of the terminal check procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
tsc$data$ptr	A POINTER to the start of the Terminal Support Code Data Area.

message$ptr	For message-passing terminal drivers, a POINTER to the message received from the controller via the RQ$RECEIVE system call. This parameter points to a structure, as follows:

```
STRUCTURE(
  data$ptr             POINTER,
  flags                WORD,
  status               WORD,
  trans$id             WORD,
  data$length          DWORD,
  forwarding$port      TOKEN,
  remote$socket        SOCKET,
  control$msg(20)      BYTE,
  reserved(4)          BYTE;
```

where

data$ptr is a POINTER to the starting address of the data portion (if any) of the received message. If the data was received in a data chain, then this parameter points to the data chain block. If this parameter is NULL, there is no optional data portion for this message.

flags is a WORD with the following encoded meaning:

Bit	Name
0-3	data$type
4-7	receive$type
8-15	reserved

where

data$type defines whether data$ptr points to a data chain (01B) or a single buffer (00B.) Other values are reserved.

receive$type is an indicator of the type of message received as follows:

Value	Message Type
0000B	Transactionless message (RQ$SEND or similar call)
0001B	Transmission or system status message
0010B	Transaction request message (RQ$SEND$RSVP or similar call)
0100B	Transaction response message (RQ$SEND$REPLY or similar call)

status contains the send message status. The status codes are

Status	Meaning
E$OK	A new message has been successfully received
E$CANCELLED	A SEND$RSVP transaction has been remotely canceled.
E$NO$LOCAL$-BUFFER	This error applies to two cases: If the receive$type parameter indicates a request message, the local port's buffer pool does not contain a buffer large enough to hold the message so the RQ$RECEIVE$FRAGMENT system call is required (message fragmentation.) If the receive$type parameter indicates a response message, the RSVP buffer supplied in the RQ$SEND$RSVP system call is not large enough to hold the response.
E$NO$REMOTE$-BUFFER	The remote port's buffer pool does not have a buffer large enough to hold the message and message fragmentation is turned off.
E$TRANSMISSION	A NACK (Negative Acknowledgment), MPC Failsafe timeout, bus or agent error, or retry expiration occurred during the transmission of the message.

trans$id is a WORD that contains the transaction ID for this message. If trans$id is zero, a new transactionless message has been received. If trans$id is not zero, it either indicates a request or response message has been received, or it indicates an asynchronous transmission status message has been received.

data$length is a DWORD that indicates the length of the data message received.

If **receive$type** indicates a newly received message, then data$length contains the length of the successfully received message.

If **receive$type** and **status** indicate request message fragmentation, the **data$length** contains the length of all the message fragments that will be received using the RQ$RECEIVE$FRAGMENT system call.

forwarding$port is a TOKEN indicates a port. The indicated port is the source port for the port that is actually receiving the message.

remote$socket is a SOCKET that has been declared as follows:

```
DECLARE socket LITERALLY STRUCTURE(
                        host$id     WORD,
                        port$id     WORD);
```

This parameter indicates the remote message source.

control$msg is the 20-byte control part of a data message.

reserved is a reserved BYTE array. Set to zero.

6.4.7 Terminal Output Procedure

The Terminal Support Code calls this procedure to display a character at a terminal connected to a nonbuffered device. The Terminal Support Code passes it the character and a pointer to the terminal's UNIT$DATA structure. If bits 6 through 8 of the TERMINAL$FLAGS field of the UNIT$DATA structure so indicate, the terminal output procedure should adjust the character's parity bit and then output the character to the terminal.

This procedure is not needed for message-passing devices and interrupt-driven buffered devices. They can send more than one output character at a time. Instead, the terminal utility procedure is used to move characters to the device's output buffer.

The syntax of the call to the user-written terminal output procedure is as follows:

```
CALL term$out(unit$data$n$p, output$character);
```

where

term$out	Name of the terminal output procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$p	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area.
output$character	BYTE containing a character that the terminal output procedure should send to the terminal.

6.4.8 Terminal Utility Procedure

The Terminal Support Code calls this procedure to perform any of several operations that apply specifically to message-passing devices and interrupt-driven buffered devices. If your device is a nonbuffered device, supply a null procedure for the terminal utility procedure.

When the Terminal Support Code calls the terminal utility procedure, it sets the FUNCTION$ID field of the unit's BUFFERED$DEVICE$DATA structure to indicate the function it wants the terminal utility procedure to perform. The function IDs and their descriptions are as follows (unless otherwise stated, the fields mentioned in the following descriptions all reside in the BUFFERED$DEVICE$DATA structure):

FUNCTION$ID Value	Description
0	The terminal utility procedure must move the number of characters specified in the OUT$COUNT field from the user's output buffer (pointed to by the USER$BUFFER$P field) to the unit's on-board output buffer. For message-passing drivers, this step involves sending a message containing the output data to the controller.
1	The Terminal Support Code has moved a number of characters (specified in the IN$COUNT field) from the unit's raw input buffer to the type-ahead buffer. If the device driver (or the device itself) is keeping track of the space remaining in the unit's input buffer, the terminal utility procedure should update its count (or send a command to the device's firmware) indicating that IN$COUNT bytes have been removed from the unit's input buffer. The driver should also decrement IN$COUNT.
2	When an input interrupt was received, the Terminal Support Code's input buffer was full. Therefore it didn't move any characters from the device's raw input buffer to the type-ahead buffer. The terminal utility procedure must send a command to the device to send the input interrupt again.
3	The modem control bit in the TERMINAL$FLAGS field of the unit's UNIT$DATA structure has changed. The terminal utility procedure should set or reset DTR according to the setting of the bit.
4	One or more of the terminal attributes that apply specifically to buffered devices have changed (in the BUFFERED$DEVICE$DATA structure, these attributes are listed in the fields from SPECIAL$MODES through SPECIAL$CHAR). The terminal utility procedure should issue controller or firmware commands to modify the device attributes to match the values listed in the BUFFERED$DEVICE$DATA structure.

FUNCTION$ID Value	Description
5	The Terminal Support Code calls this function to find out the amount of space available in the unit's output buffer. When this function is called, the terminal utility procedure must indicate how much room is left in the output buffer for additional characters by placing the number of bytes in the UNITS$AVAILABLE field.
6	Output has been cancelled, or the Terminal Support Code has received a discard output control character (normally Control-O). The terminal utility procedure must clear the unit's output buffer.
7	The Terminal Support Code has received an output control character that changes the output state of the terminal. The terminal utility procedure must examine the BUFF$OUTPUT$STATE field and set the output state accordingly. For example, if an operator types a CONTROL-S, the Terminal Support Code sets bit 1 in the BUFF$OUTPUT$STATE field to 1. In this case, the terminal utility procedure must stop output to the terminal.
8	Characters must be echoed to the terminal. The terminal utility procedure must move the number of characters specified in ECHO$COUNT from the buffer pointed to by ECHO$BUFFER$P to the unit's on-board output buffer. Any characters that the terminal utility procedure doesn't move are lost. For message-passing drivers, this step involves sending a message containing these characters to the controller.
9	Input has been cancelled. The terminal utility procedure must clear the unit's raw input buffer and set RAW$OUT equal to RAW$IN.
0AH	The terminal utility procedure must update the RAW$IN field of the UNIT$DATA structure to the correct value.
0BH, 0CH	Intel reserves these fields for future use. Drivers do not set these values.
0DH	If your controller does not automatically send output interrupts, the driver must request the controller to send an interrupt/message when the output buffer on the controller is empty. The driver must then indicate an output interrupt to the Terminal Support Code. If your controller automatically sends response interrupts/messages when an output request is completed, ignore this function code.

The syntax of a call to the user-written terminal utility procedure is as follows:

```
CALL term$utility(unit$data$n$p);
```

where

unit$data$n$p	POINTER to the terminal's UNIT$DATA structure in the Terminal Support Code Data Area.

6.5 PROCEDURES' USE OF DATA STRUCTURES

Table 6-1 helps you sort out the responsibilities of the various procedures in a terminal device driver. In the table, the following codes refer to those procedures:

(1) terminal initialization

(2) terminal finish

(3) terminal setup

(4) terminal answer

(5) terminal hangup

(6) terminal check

(7) terminal output

(8) terminal utility

Also, "System" and "ICU" are used in Table 6-1 to indicate the Extended iRMX II software and the Extended iRMX II Interactive Configuration Utility, respectively. In addition, the numbers following immediately after "Term$flags" are bit numbers in that word.

Table 6-1. Uses of Fields in Terminal Driver Data Structures (continued)

	Filled in/Changed by	Can or Will be Read by
TSC$DATA		
IOS$DATA$SEGMENT	System	(1)-(7)
STATUS	(1),(3)	System
INTERRUPT$TYPE	(6)	System
INTERRUPTING$UNIT	(6)	System
DEV$INFO$PTR	System	(1)-(7)
USER$DATA$PTR	System	(1)-(7)
UNIT$DATA		
UNIT$INFO$PTR	System,ICU	System
TERMINAL$FLAGS (0-2)	System,ICU	System
TERMINAL$FLAGS (3)	System,ICU	(3),(8)
TERMINAL$FLAGS (4-5)	System,ICU	(3),(6)
TERMINAL$FLAGS (6-8)	System,ICU	(3),(6),(7)
IN$RATE	System,ICU,(3),(6)	(3)
OUT$RATE	System,ICU	(3)
SCROLL$NUMBER	System,ICU	System
X$Y$SIZE	System	System
X$Y$OFFSET	System	System
RAW$SIZE	(1)	System,(6)
RAW$DATA$P	(1)	System,(6)
RAW$IN	(1),(8)	System,(6)
RAW$OUT	(1),(8) System	System,(6)
OUTPUT$SCROLL$COUNT	System,(8)	System,(7)
UNIT$NUMBER	System	System,(1)
BUFFERED$DEVICE$DATA	(3)	System,(3),(8)
BUFFERED$DEVICE$DATA		
BUFFERED$DEVICE	(1),(3)	System
BUFF$INPUT$STATE	System,(8)	(8)
BUFF$OUTPUT$STATE	System,(8)	(8)
SELECT	(1)	(8)
LINE$RAM$P	(1)	System,(1),(3),(8)
FUNCTION$ID	System	(8)
IN$COUNT	System	(8)
OUT$COUNT	System,(8)	(8)
UNITS$AVAILABLE	(8)	System
OUTPUT$BUFFER$SIZE	(1)	System,(8)
USER$BUFFER$P	System	System,(8)
ECHO$COUNT	System	(8)
ECHO$BUFFER$P	System	(8)
RECEIVED$SPECIAL	(6)	System,(6)

Table 6-1. Uses of Fields in Terminal Driver Data Structures

	Filled in/Changed by	Can or Will be Read by
SPECIAL$MODES	System	System,(6),(8)
HIGH$WATER$MARK	System	(3)
LOW$WATER$MARK	System	(3)
FC$ON$CHAR	System	(3)
FC$OFF$CHAR	System	(3)
LINK$PARAMETER	System	(3)
SPC$HI$WATER$MARK	System	(8)
SPECIAL$CHAR	System	(8)
TERMINAL$DEVICE$INFORMATION		
NUM$UNITS	ICU	System
DRIVER$DATA$SIZE	ICU	System
STACK$SIZE	ICU	System
TERM$INIT	ICU	System
TERM$FINISH	ICU	System
TERM$SETUP	ICU	System
TERM$OUT	ICU	System
TERM$ANSWER	ICU	System
TERM$HANGUP	ICU	System
TERM$UTILITY	ICU	System
INTERRUPTS		
INTERRUPT$LEVEL	ICU	System
TERM$CHECK	ICU	System
DRIVER$INFO	ICU	(1)-(7)

CHAPTER 7 WRITING A CUSTOM DRIVER

Custom device drivers are drivers you create in their entirety because your device doesn't fit into the common, random access, or terminal device category, either because the device requires a priority-ordered queue, multiple interrupt levels, or because of some other reasons that you have determined. When you write a custom device driver, you must provide all of the features of the driver, including creating and deleting resources, implementing a request queue, and creating an interrupt handler. You can do this in any manner that you choose as long as you supply the following four procedures for the I/O System to call:

An Initialize I/O Procedure. This procedure must initialize the device and create any resources needed by the procedures in the driver.

A Finish I/O Procedure. This procedure must perform any final processing on the device and delete resources created by the remainder of the procedures in the driver.

A Queue I/O Procedure. This procedure must place the I/O requests on a queue of some sort, so that the device can process them when it becomes available.

A Cancel I/O Procedure. This procedure must cancel a previously queued I/O request.

For the I/O System to communicate with your device driver procedures, you must place the addresses of these four procedures in the DUIBs that correspond to the units of the device.

The next four sections describe the format of each of the I/O System calls to these four procedures. Your procedures must conform to these formats.

7.1 INITIALIZE I/O PROCEDURE

The I/O System calls the Initialize I/O procedure when an application task makes an RQ$A$PHYSICAL$ATTACH$DEVICE system call and no units of the device are currently attached.

The Initialize I/O procedure must perform any initial processing necessary for the device or the driver. If the device requires an interrupt task, region, or device data area, the Initialize I/O procedure should create them.

The format of the call to the Initialize I/O procedure is as follows:

```
CALL init$io(duib$p, ddata$p, status$p);
```

where

init$io	Name of the Initialize I/O procedure. You can use any name for this procedure as long as it does not conflict with other procedure names. You must, however, provide its starting address in the DUIBs of all device-units that it services.
duib$p	POINTER to the DUIB of the device-unit for which the request is intended. This is an input parameter supplied by the I/O System. The init$io procedure uses this DUIB to determine the characteristics of the unit.
ddata$p	POINTER to a TOKEN in which the init$io procedure can place the location of a data storage area, if the device driver needs such an area. If the device driver requires that a data area be associated with a device (to contain the head of the I/O queue, DUIB addresses, or status information), the init$io procedure should create this area and save its location via this pointer. If the driver does not need such a data area, the init$io procedure should return a NIL pointer in this variable.
status$p	POINTER to a WORD in which the init$io procedure must place the status of the initialize operation. If the operation is completed successfully, the init$io procedure must return the E$OK condition code. Otherwise it should return the appropriate exception code. If the init$io procedure does not return the E$OK condition code, it must delete any resources that it has created.

7.2 FINISH I/O PROCEDURE

The I/O System calls the Finish I/O procedure after an application task makes an RQ$A$PHYSICAL$DETACH$DEVICE system call to detach the last unit of a device.

The Finish I/O procedure performs any necessary final processing on the device. It must delete all resources created by other procedures in the device driver and must perform final processing on the device itself, if the device requires such processing.

The format of the call to the Finish I/O procedure is as follows:

```
CALL finish$io(duib$p, ddata$t);
```

where

finish$io	Name of the Finish I/O procedure. You can specify any name for this procedure as long as it does not conflict with other procedure names. You must, however, place its starting address in the DUIBs of all device-units that it services.

duib$p	POINTER to the DUIB of the device-unit of the device being detached. This is an input parameter supplied by the I/O System. The finish$io procedure needs this DUIB to determine the device on which to perform the final processing.
ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure (or SELECTOR$OF(NIL), if none was created). This is an input parameter supplied by the I/O System. The finish$io procedure must delete this resource and any others created by driver routines.

7.3 QUEUE I/O PROCEDURE

The I/O System calls the Queue I/O procedure to place an I/O request on a queue, so that it can be processed when the device is not busy. The Queue I/O procedure must actually start the processing of the next I/O request on the queue if the device is not busy. The format of the call to the Queue I/O procedure is as follows:

```
CALL queue$io(iors$t, duib$p, ddata$t);
```

where

queue$io	Name of the Queue I/O procedure. You can use any name for this procedure as long as it does not conflict with other procedure names. You must, however, place its starting address in the DUIBs of all device-units that it services.
iors$t	SELECTOR containing the location of an IORS. This is an input parameter supplied by the I/O System. The IORS describes the request. When the request is processed, the driver must fill in the status fields and send the IORS to the response mailbox indicated in the IORS. Chapter 4 describes the format of the IORS. It lists the information that the I/O System supplies when it passes the IORS to the queue$io procedure and indicates the fields of the IORS that the device driver must fill in.
duib$p	POINTER to the DUIB of the device-unit for which the request is intended. This is an input parameter supplied by the I/O System.
ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure (or SELECTOR$OF(NIL), if none was created). This is an input parameter supplied by the I/O System. The queue$io procedure can place any necessary information in this area to update the request queue or status fields.

7.4 CANCEL I/O PROCEDURE

The I/O System can call the Cancel I/O procedure to cancel one or more previously queued I/O requests. It calls Cancel I/O under any of the following conditions:

- If the user invokes an RQ$A$PHYSICAL$DETACH$DEVICE system call with a hard detach option (refer to the *Extended iRMX II Basic I/O System Calls* manual for a description of this call). This system call forcibly detaches all objects associated with a device-unit.
- If the job containing the task which made an I/O request is deleted. The I/O System calls the Cancel I/O procedure to remove any requests that tasks in the deleted job might have made.
- If the user deletes a connection to a device. The I/O System calls Cancel I/O to remove any I/O requests pending for the device.

If the device cannot guarantee a request will be finished within a fixed amount of time (such as waiting for input from a terminal keyboard when the operator may or may not supply that input), the Cancel I/O procedure must actually stop the device from processing the request. If the device guarantees that all requests finish in an acceptable amount of time, the Cancel I/O procedure does not have to stop the device itself, but only removes requests from the queue.

The format of the call to the Cancel I/O procedure is as follows:

```
CALL cancel$io(cancel$id, duib$p, ddata$t);
```

where

cancel$io	Name of the Cancel I/O procedure. You can use any name for this procedure as long as it doesn't conflict with other procedure names. You must, however, place its starting address in the DUIBs of all device-units that it services.
cancel$id	WORD containing the ID value for the I/O requests that are are to be canceled. This is an input parameter supplied by the I/O System. Any pending requests with this ID value in the cancel$id field of their IORSs must be removed from the queue of requests by the Cancel I/O procedure. Moreover, the I/O System places a CLOSE request with the same cancel$id value in the queue. The CLOSE request must not be processed until all other requests with that cancel$id value have been removed from the queue.
duib$p	POINTER to the DUIB of the device-unit for which the request cancellation is intended. This is an input parameter supplied by the I/O System.

ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure (or SELECTOR$OF(NIL), if none was created). This is an input parameter supplied by the I/O System. This data storage area may contain the request queue.

7.5 IMPLEMENTING A REQUEST QUEUE

Making I/O requests via system calls and the actual processing of these requests by I/O devices are asynchronous activities. When a device is processing one request, many more can be accumulating. Unless the device driver has a mechanism for placing I/O requests on a queue of some sort, these requests will become lost. For common and random access devices, the I/O System's support code forms this queue by creating a doubly-linked list. The list is used by the QUEUE$IO and CANCEL$IO procedures, as well as by INTERRUPT$TASK.

Using this mechanism of the doubly linked list, the common and random access support code implements a FIFO queue for I/O requests. If you are writing a custom device driver, you might want to take advantage of the LINK$FOR and LINK$BACK fields that are provided in the IORS and implement a scheme similar to the following for queuing I/O requests.

Each time a user makes an I/O request, the I/O System passes an IORS for this request to the device driver, in particular to the Queue I/O procedure of the device driver. The Queue I/O procedure for common and random access drivers makes use of the LINK$FOR and LINK$BACK fields of the IORS to link this IORS together with IORSs for other requests that have not yet been processed.

This queue is set up in the following manner. The device driver routine that actually sends data to the controller accesses the first IORS on the queue. The LINK$FOR field in this IORS points to the next IORS on the queue. The LINK$FOR field in the second IORS points to the third IORS on the queue, and so forth until, in the last IORS on the queue, the LINK$FOR field points back to the first IORS on the queue. The LINK$BACK fields operate in the same manner. The LINK$BACK field of the last IORS on the queue points to the previous IORS. The LINK$BACK field of the second to last IORS points to the third to last IORS on the queue, and so forth, until, in the first IORS on the queue, the LINK$BACK field points to the last IORS in the queue. This kind of queue is illustrated in Figure 7-1.

The device driver can add or remove requests from the queue by adjusting LINK$FOR and LINK$BACK pointers in the IORSs.

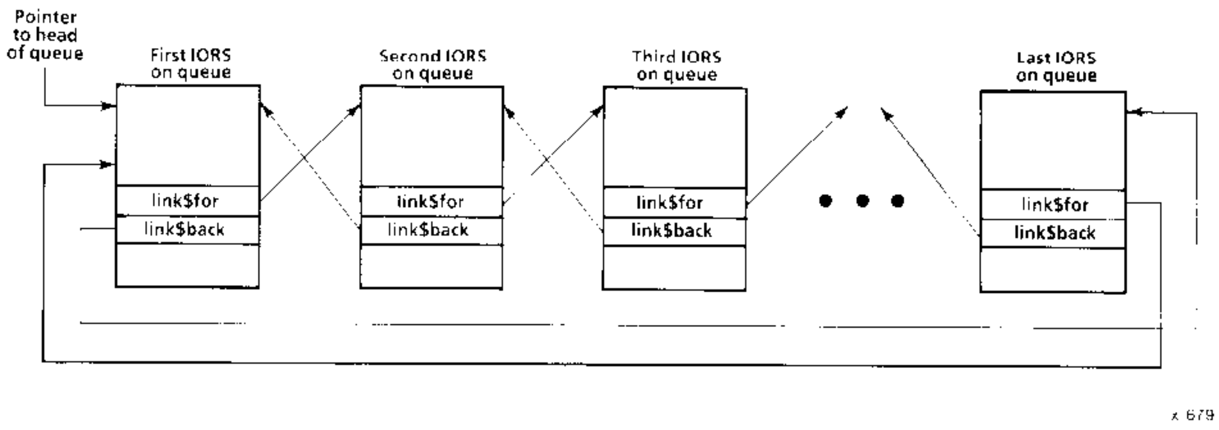


Figure 7-1. Request Queue

To handle the dual problems of locating the queue and ascertaining whether the queue is empty, you can use a variable such as head$queue. If the queue is empty, head$queue contains the value SELECTOR$OF(NIL). Otherwise, head$queue contains the token for the first IORS in the queue.

CHAPTER 8
HANDLING I/O REQUESTS

To perform I/O operations, tasks invoke Basic or Extended I/O System calls. This chapter outlines the two basic steps involved in processing these system calls: which device driver procedures the I/O System calls, and what operations the device driver must perform after being called.

8.1 I/O SYSTEM RESPONSES TO I/O SYSTEM CALLS

This section shows which device driver procedures the I/O System calls when it processes each of the I/O System calls. When the I/O System calls multiple driver procedures, the order of the calls is significant.

8.1.1 Attach Device System Calls

For attach device system calls, the I/O System calls a different set of procedures depending on whether other units of the device have already been attached.

When the I/O System receives the first attach device system call for a device, it calls the initialize I/O procedure to initialize the device as a whole and create the device data storage area and interrupt task(s). Then the I/O System calls the queue I/O procedure, with the FUNCT field of the IORS set to F$ATTACH (4).

When the I/O System receives an attach device request system call that is not the first for the device, it omits the call to Initialize I/O, but still calls Queue I/O with the FUNCT field of the IORS set to F$ATTACH (4).

8.1.2 Detach Device System Calls

As with attach device system calls, circumstances dictate which procedures the I/O System calls to detach a device.

If there is more than one unit of the device attached when the I/O System receives the detach device request, the I/O System calls the queue I/O procedure, with the FUNCT field of the IORS set to F$DETACH (5). The queue I/O procedure performs cleanup operations on the selected unit, if necessary.

If there is only one attached unit on the device when the I/O System receives the detach device system call, the I/O System calls two procedures. First it calls Queue I/O, as in the previous case. Then it calls the finish I/O procedure to perform cleanup operations for the device as a whole (if necessary) and to delete any objects created by the initialize I/O procedure.

8.1.3 Read, Write, Open, Close, Seek, and Special System Calls

When the I/O System receives a read, write, open, close, seek, or special system call, it sets the FUNCT field of the IORS appropriately and calls the queue I/O procedure to handle the operation. The "Actions Required of the Device Driver" section of this chapter explains the actions the queue I/O procedure must take.

8.1.4 Cancel Requests

When a connection is deleted while I/O is in progress, such as when a job is deleted, the I/O System calls two device driver procedures.

First it calls the cancel I/O procedure to remove from the request queue all requests that contain the same Cancel ID value as that in the current request (the I/O System fills in the CANCEL$ID field of the IORS with the Cancel ID value). The cancel I/O procedure stops the processing of the current request, if necessary.

Then the I/O System calls the queue I/O procedure with the FUNCT field of the IORS set to F$CLOSE (7). When this request reaches the front of the queue, it is simply returned to the indicated response mailbox.

8.2 ACTIONS REQUIRED OF THE DEVICE DRIVER

The I/O System can make eight types of requests of a device driver. One of these requests (SPECIAL) has 14 separate subrequests associated with it. The I/O System identifies the kind of request by setting the FUNCT field (and for SPECIAL requests, the SUBFUNCT field) of the IORS. Then it passes the IORS to the device driver. This chapter summarizes the actions required of a device driver whenever it receives any of these requests or subrequests. Unless otherwise specified, all the actions listed in this chapter must be performed by the Queue I/O procedure or a procedure it calls.

If a driver does not support a particular function or subfunction, it must place the E$IDDR exception code in the IORS.STATUS field before returning.

As Chapter 6 mentions, the Terminal Support Code does not pass IORSs to the terminal driver procedures. However, if you write a custom driver for a terminal, your driver must process all IORSs directly. Random access and common drivers must also process most IORSs (the random access support code processes a few itself). Unless otherwise noted, the following sections assume that your driver (either custom, random access, or common) handles all the actions described.

8.2.1 F$READ--Function Code 0

The device driver must take the following actions to support F$READ requests:

1. Use the value in IORS.COUNT to determine the number of bytes to read from the device.
2. Read the bytes from the location specified in IORS.DEV$LOC. The location is specified either as an absolute byte count, an absolute sector number, or as the track and sector numbers. If the device is a flexible diskette drive formatted in the Intel standard format, calculate the real location after accounting for the special formatting on track 0 (refer to Appendix C). Read the data into the memory pointed to by the IORS.BUFF$P pointer.

   The IORS.DEV$LOC field is not used by drivers for terminal devices, nor by common drivers such as tape drivers.
3. Indicate the number of bytes read by placing that number into the IORS.ACTUAL field. If no error occurred, this value should be the same as the value in the IORS.COUNT field. If an error occurred, the IORS.ACTUAL value will be less.
4. Place the read status into IORS.STATUS. This status should be E$OK (0) if the operation completed successfully. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.2 F$WRITE--Function Code 1

The device driver must take the following actions to support F$WRITE requests:

1. Use the value in IORS.COUNT to determine the number of bytes to write to the device.
2. Read the bytes from the area of memory pointed to by the IORS.BUFF$P pointer.
3. Write the bytes to the location specified in IORS.DEV$LOC. The location is specified either as an absolute byte count, an absolute sector number, or as the track and sector numbers. If the device is a flexible diskette drive formatted in the Intel standard format, calculate the real location after accounting for the special formatting on track 0 (refer to Appendix C).

   The IORS.DEV$LOC field is not used by drivers for terminal devices, nor by common drivers such as tape drivers.

4. Indicate the number of bytes written by placing that number into the IORS.ACTUAL field. If no error occurred, this value should be the same as the value in the IORS.COUNT field. If an error occurred, the IORS.ACTUAL value will be less.
5. Place the write status into IORS.STATUS. This status should be E$OK (0) if the operation completed successfully. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.3 F$SEEK--Function Code 2

The device driver must take the following actions to support F$SEEK requests:

1. Move the device's head to the location specified in IORS.DEV$LOC. The location is specified either as an absolute byte count, an absolute sector number, or as the track and sector numbers. If the device is a flexible diskette drive formatted in the Intel standard format, calculate the real location after accounting for the special formatting on track 0 (refer to Appendix C).
2. Place the seek status into IORS.STATUS. This status should be E$OK (0) if the operation completed successfully. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.4 F$ATTACH--Function Code 4

The device driver must take the following actions to support F$ATTACH requests:

1. Initialize the unit specified in the IORS.UNIT field. Also initialize any driver data structures that are specific to that unit.
2. Set the IORS.STATUS field to E$OK (0) if the operation completed successfully. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.5 F$DETACH--Function Code 5

The device driver must take the following actions to support F$DETACH requests:

1. Delete any driver data structures created by the device driver that are specific to the unit listed in IORS.UNIT.
2. Set the IORS.STATUS field to E$OK (0) if the operation completed successfully. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.6 F$OPEN--Function Code 6

The device driver must take the following actions to support F$OPEN requests:

1. Prepare the unit for accessing a file. Usually, no processing is involved for this operation.
2. Place the open status into IORS.STATUS. This status should be E$OK (0) if the operation completed successfully. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.7 F$CLOSE--Function Code 7

The device driver must take the following actions to support F$CLOSE requests:

1. Prepare the unit for closing a file. Usually, no processing is involved for this operation.
2. Place the close status into IORS.STATUS. This status should be E$OK (0) if the operation completed successfully. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.8 F$SPECIAL--Function Code 3

The device driver must take the following actions to support F$SPECIAL requests:

Examine the IORS.SUBFUNCT field to determine the action to take. Intel reserves subfunction numbers 0 through 32,767. User-defined subfunctions can have subfunction numbers from 32,768 through 65,535 and can be used with the physical file driver only.

Most subfunctions use auxiliary information pointed to by the IORS.AUX$P pointer. The format of this information depends on the subfunction invoked. The following paragraphs describe the actions of the driver for each subfunction.

8.2.8.1 FS$FORMAT$TRACK--Subfunction 0

If the unit is a tape drive, perform the following steps:

1. Rewind the tape to BOT (beginning of tape).
2. Erase the entire tape.
3. Rewind the tape again.

If the unit is a disk drive, format a track according to the information pointed to by the IORS.AUX$P pointer. This information has the following structure:

```
DECLARE FORMAT$TRACK STRUCTURE(
    TRACK$NUMBER     WORD,
    INTERLEAVE       WORD,
    TRACK$OFFSET     WORD,
    FILL$CHAR        BYTE);
```

Use the following procedure to format the disk:

1. If the value in the TRACK$NUMBER field of the FORMAT$TRACK structure is greater than the highest track on the disk, set the IORS.STATUS field to the E$SPACE exception code.
2. If the value in the TRACK$NUMBER field is valid, format the track using the INTERLEAVE and FILL$CHAR values from the FORMAT$TRACK structure, and using the device characteristics listed in the DUIB (DEV$GRAN and FLAGS). If necessary, also use the device-specific characteristics listed in the unit information table.
3. If the drive includes information about the bad sectors or bad tracks on the drive, retrieve this information and assign alternate sectors or an alternate track for the track listed in the FORMAT$TRACK structure. Depending on how the driver works, it might not need to retrieve the data more than once. But it should check to assign alternate sectors or an alternate track each time it formats a track. Refer to Appendix D for information about the location and format of the bad sector information.
4. If this is a flexible diskette drive and bit 4 of the FLAGS field in the DUIB is set to zero (indicating standard formatting), track 0 must be formatted differently. Refer to Appendix C for information about the standard diskette format.
5. Place the format status in the IORS.STATUS field. Set this field to E$OK (0) if no errors occur. If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.8.2 FS$QUERY--Subfunction 0

This is a stream file operation handled totally by the I/O System's stream file driver.

8.2.8.3 FS$SATISFY--Subfunction 1

This is a stream file operation handled totally by the I/O System's stream file driver.

8.2.8.4 FS$NOTIFY--Subfunction 2

For this subfunction, the IORS.AUX$P pointer points to the following structure:

```
DECLARE SETUP$NOTIFY STRUCTURE(
    MAILBOX     TOKEN,
    OBJECT      TOKEN);
```

The random access support code handles FS$NOTIFY requests for random access and common drivers. If the driver is a custom driver, it must perform the following steps:

1. Save the parameters passed in the SETUP$NOTIFY structure in variables for later use.
2. Whenever a media change occurs, such as opening a diskette drive door or removing a tape cartridge (these usually cause an interrupt that the driver can identify as a media-change interrupt), the driver must send the SETUP$NOTIFY.OBJECT token to the SETUP$NOTIFY.MAILBOX mailbox.

If the driver is a random access driver, the I/O System doesn't pass the FS$NOTIFY request to the user-written driver code. However, the driver must call the I/O System-supplied NOTIFY procedure whenever it detects a media change. This procedure sends the object to the mailbox for the driver.

8.2.8.5 FS$GET$DRIVE$DATA--Subfunction 3

1. Copy the disk drive or tape drive characteristics (as obtained from the DUIB, Device Information Table, Unit Information Table, or the device itself) into the structure pointed to by IORS.AUX$P. For disk drives, the structure is as follows:

   ```
   DECLARE DISK$DRIVE$DATA STRUCTURE(
       CYLINDERS       WORD,
       FIXED           BYTE,
       REMOVABLE       BYTE,
       SECTORS         BYTE,
       SECTOR$SIZE     WORD,
       ALTERNATES      BYTE);
   ```

   For tape drives, the structure is as follows:

   ```
   DECLARE TAPE$DRIVE$DATA STRUCTURE(
       TAPE            BYTE,
       RESERVED(7)     BYTE);
   ```

   Refer to the description of the A$SPECIAL system call in the *Extended iRMX II Basic I/O System Calls* manual for information about the contents of these structures.

2. If the operation completed successfully, set the IORS.STATUS field to E$OK (0). If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.8.6 FS$GET$TERMINAL$ATTRIBUTES--Subfunction 4

For terminal drivers, the Terminal Support Code performs this operation without passing it on to the user-written driver code. Random access and common drivers do not support this operation and should set the IORS.STATUS field to indicate the E$IDDR exception code.

If custom terminal drivers support this subfunction, they should place information about the terminal in the structure pointed to by the IORS.AUX$P pointer. The format of the structure that other terminal drivers use is as follows:

```
DECLARE TERMINAL$ATTRIBUTES STRUCTURE(
    NUM$WORDS           WORD,
    NUM$USED            WORD,
    CONNECTION$FLAGS    WORD,
    TERMINAL$FLAGS      WORD,
    IN$BAUD$RATE        WORD,
    OUT$BAUD$RATE       WORD,
    SCROLL$LINES        WORD,
    X$Y$SIZE            WORD,
    X$Y$OFFSET          WORD,
    SPECIAL$MODES       WORD,
    HIGH$WATER$MARK     WORD,
    LOW$WATER$MARK      WORD,
    FC$ON$CHAR          WORD,
    FC$OFF$CHAR         WORD,
    LINK$PARAMETER      WORD,
    SPC$HI$WATER$MARK   WORD,
    SPECIAL$CHAR(4)     BYTE);
```

Refer to Chapter 4 and to the *Extended iRMX II Basic I/O System Calls* manual for more information about the fields in this structure.

8.2.8.7 FS$SET$TERMINAL$ATTRIBUTES--Subfunction 5

For terminal drivers, the Terminal Support Code places the terminal attributes in a TERMINAL$ATTRIBUTES structure that is pointed to by the IORS.AUX$P pointer. This is the same structure used by FS$GET$TERMINAL$ATTRIBUTES. The Terminal Support Code calls the terminal check procedure changes in baud rate and parity checking. It calls the terminal utility procedure for changes in those attributes that apply specifically to buffered devices (the SPECIAL$MODES through SPECIAL$CHAR fields of the structure). The device driver procedure that receives control must examine the structure and ensure that the device is set up with the corresponding attributes.

Random access and common drivers do not support this operation and should set the IORS.STATUS field to indicate the E$IDDR exception code.

If custom terminal drivers support this subfunction, they should examine the structure pointed to by the IORS.AUX$P pointer and act on the changes. Otherwise, they should return an E$IDDR exception code in the IORS.STATUS field.

8.2.8.8 FS$SET$SIGNAL--Subfunction 6

For terminal drivers, the Terminal Support Code performs this operation without passing it on to the user-written driver code. Random access and common drivers do not support this operation and should set the IORS.STATUS field to indicate the E$IDDR exception code.

For custom terminal drivers, the IORS.AUX$P pointer points to a structure of the following format:

```
DECLARE SIGNAL$CHARACTER STRUCTURE(
    SEMAPHORE    TOKEN,
    CHARACTER    BYTE);
```

To be compatible with the Terminal Support Code (and thus allow the Human Interface CONTROL-C mechanism to operate properly), the driver must perform the following operations. Otherwise, the driver can set up its own interpretation of signal characters.

1. Save the parameters passed in the SIGNAL$CHARACTER structure in driver variables for later use. The driver should accept SIGNAL$CHARACTER.CHARACTER values in the range of 0 through 31 decimal or 32 through 63 decimal.

   - If the value is in the range of 0 through 31 decimal, it is the ASCII code of the signal character.

   - If the value is in the range of 32 through 63 decimal, the driver must subtract 32 decimal from the value to obtain the ASCII code of the signal character. These higher values indicate that the driver must flush the terminal's input buffer when it receives the signal character.

   - If the value is greater than 63 decimal, the driver can ignore the FS$SET$SIGNAL request.

2. Whenever the character indicated in the SIGNAL$CHARACTER.CHARACTER field is entered at the terminal, send a unit to the semaphore listed in SIGNAL$CHARACTER.SEMAPHORE. If the signal character was originally specified in the range 32 through 63 decimal, also flush the terminal's input buffer.

If the driver doesn't support this subfunction, it should return an E$IDDR exception code in the IORS.STATUS field.

8.2.8.9 FS$RESET–Subfunction 7

1. If the unit is a tape drive, rewind the tape.

If the unit is a disk drive, recalibrate the disk (move the head or heads to track 0).

2. If the operation completes successfully, set the IORS.STATUS field to E$OK (0). If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.8.10 FS$READ$FILE$MARK–Subfunction 8

If the unit is a tape unit, perform the following steps:

1. Move the tape to the next file mark.
2. If the operation completes successfully, set the IORS.STATUS field to E$OK (0). If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

If the unit is not a tape drive, place an E$IDDR exception code in the IORS.STATUS field and return.

8.2.8.11 FS$WRITE$FILE$MARK–Subfunction 9

If the unit is a tape unit, perform the following steps:

1. Write a file mark on the tape at the current tape position.
2. If the operation completes successfully, set the IORS.STATUS field to E$OK (0). If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

If the unit is not a tape drive, place an E$IDDR exception code in the IORS.STATUS field and return.

8.2.8.12 FS$RETENSION$TAPE–Subfunction 10

If the IORS.UNIT field indicates that this is a tape unit, perform the following steps to ensure that the tape is wound evenly and is straight in the cartridge:

1. Rewind the tape.
2. Fast forward the tape to the end.
3. Rewind the tape again.
4. If the operation completes successfully, set the IORS.STATUS field to E$OK (0). If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

If the unit is not a tape drive, place an E$IDDR exception code in the IORS.STATUS field and return.

8.2.8.13 FS$SET$BAD$INFO–Subfunction 12

1. Examine the DEV$GRAN field of the DUIB to determine the sector size of the device.
2. Based on the sector size, move the head to the appropriate surface of the second-to-last cylinder, as follows:

128-byte sectors	last surface
256-byte sectors	last surface-1
512-byte sectors	last surface-2
1024-byte sectors	last surface-3

   See Appendix D for more information.
3. Format the entire track.
4. Write the value 0ABCDH in the first word of the track. Then write the information from the BAD$TRACK$INFO structure (beginning with the COUNT field) to the track. Write the entire bad-track information four times, as shown in Appendix D.
5. If the operation completes successfully, set the IORS.STATUS field to E$OK (0). If an error occurred, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

8.2.8.14 FS$GET$BAD$INFO--Subfunction 13

1. Examine the DEV$GRAN field of the DUIB to determine the sector size of the device.
2. Based on the sector size, move the head to the appropriate surface of the second-to-last cylinder, as follows:

128-byte sectors	last surface
256-byte sectors	last surface-1
512-byte sectors	last surface-2
1024-byte sectors	last surface-3

   See Appendix D for more information.
3. Read the bad-track information into the BAD$TRACK$INFO structure. The format of the information is shown in Appendix D.
4. If the read operation completes successfully, set the IORS.STATUS field to E$OK (0). If an I/O error occurred, attempt to read the next copy of the bad-track information. If I/O errors occur when reading all four copies of the information, place the general exception code into IORS.STATUS and a specific error code into IORS.UNIT$STATUS.

CHAPTER 9 CONFIGURING A USER-WRITTEN DEVICE DRIVER

You can write the modules for your device driver in either PL/M-286 or in the ASM286 Macro Assembly Language. However, you must adhere to the following guidelines:

- If you use PL/M-286, you must define your routines as REENTRANT, PUBLIC procedures, and compile them using the ROM and COMPACT controls.
- If you use assembly language, your routines must follow the conditions and conventions used by the PL/M-286 COMPACT size control. In particular, your routines must function in the same manner as reentrant PL/M-286 procedures with the ROM and COMPACT controls set. The *ASM286 Macro Assembler Operating Instructions* manual describes these conditions and conventions.

There are two ways to use the iRMX II Interactive Configuration Utility to configure a user-written driver into your application system. One way is to use the Intel-supplied tools UDS and ICUMRG to modify the ICU so that it supports your device driver automatically. The second way is to add your driver as a custom driver, without first modifying the ICU. Whichever method you use, you must perform these operations first:

- For each device driver that you have written, assemble or compile the code for the driver.
- Put the resulting object modules for terminal drivers in a single library, such as TERMINAL.LIB.
- Put the resulting object modules for random/common/custom drivers in a single module, such as DRIVER.LIB.

9.1 ADDING DRIVERS WITH THE UDS AND ICUMRG UTILITIES

The iRMX II package contains two utilities that help you add support for user-written device drivers to the Interactive Configuration Utility (ICU). With these utilities, you can add screens to the ICU so that configuring your device driver is simply a matter of running the ICU and answering the appropriate questions. You can add device information, unit information, and device-unit information screens for as many user-written device drivers as you wish.

The two utilities are UDS (User Device Support) and ICUMRG (ICU Merge). The UDS utility transforms files of screen specifications into files that are compatible with the ICU. ICUMRG merges these new files into the ICU. Figure 9-1 is a flow chart that gives an overview of using these utilities. The following sections describe the utilities in detail. These sections assume that you are familiar with the general operation of the ICU. If you are not familiar with the ICU, refer to the *Guide To Using The Extended iRMX II Interactive Configuration Utility* or the *Extended iRMX II Interactive Configuration Utility Reference Manual* for more information.

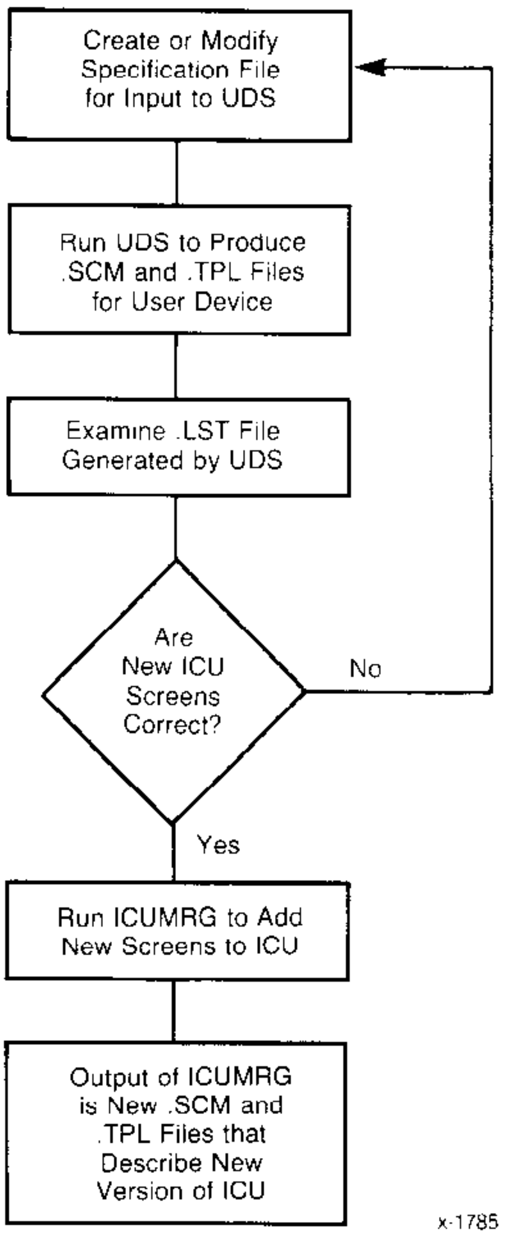

Create or Modify Specification File for Input to UDS
Run UDS to Produce .SCM and .TPL Files for User Device
Examine .LST File Generated by UDS
Are New ICU Screens Correct?
No
Yes
Run ICUMRG to Add New Screens to ICU
Output of ICUMRG is New .SCM and .TPL Files that Describe New Version of ICU
x-1785

Figure 9-1. Adding Drivers with UDS and ICUMRG

9.1.1 UDS UTILITY

The UDS utility lets you set up a device information screen, a unit information screen, and a device-unit information screens for your user-written driver. You set up these screens by placing information in a file that UDS reads. When you set up a screen, you choose from a set of standard screens. For example, when describing a device information screen, you can choose from three terminal support screens, two random access support screens, and a general screen.

Auxiliary lines can also be added to the device information and unit information screens. This allows your device-specific information to be entered during configuration. By choosing the appropriate screens and adding the correct number of auxiliary lines, you can set up the ICU to handle the configuration of almost any device driver. Depending on the number of auxiliary fields defined, you can provide the new auxiliary fields with descriptive names.

Using the input file you provide, the UDS utility creates two files that define the new screens you specified. These files have extensions .SCM and .TPL. Once these files are created, the ICUMRG utility can be used to merge these files with the ICU. In addition to the .SCM and .TPL files, the UDS produces a listing file that has a .LST extension. The list file shows how the screens will look when added the the ICU.

9.1.1.1 UDS Input File

Before invoking the UDS utility, you must create an input file that defines how the ICU screens for your device driver should look. Figure 9-2 shows the format of that input file. The information in brackets ([]) is there just to describe the lines of the file; it is not part of the input file. The "xxxx" characters indicate that you must fill in a value there.

Included with the UDS are two input file templates: **TEMPLATE_1** and **TEMPLATE_2.** These files contain example UDS user input files. **TEMPLATE_1.UDS** is a basic file, and contains no auxiliary help fields. **TEMPLATE_2.UDS** is a complete input file, and contains examples of most auxiliary fields. You can modify these files to suit your individual needs.

```
#version = xxxx             [1-4 character version number]
#name = xxxx                [1-25 character name]
#abbr = xxx                 [1-3 character abbreviation]
#driver = x                 [driver type value, from 1 to 7]
#device                     [start of device information]
#dev_aux = xx               [number of auxiliaries, from 0 to 20]
#d01 = 'parameter name'     [1-41 character parameter name, in quotes]
d01 help information        [0-1024 character help information]
                            A maximum of 1024 chars, help msgs are required
   .
   .                        [names and help information for other]
   .                        [auxiliary parameters]

#end                        [end of device information]
#unit                       [start of unit information]
#unit_aux = xx              [number of auxiliaries, from 0 to 20]
#u01 = 'parameter name'     [1-41 character parameter name, in quotes]
#u01 help information        [0-1024 character help information]
                            A maximum of 1024 chars, help msgs are required
   .
   .                        [names and help information for other]
   .                        [auxiliary parameters]

#end                        [end of unit information]
#duib
#duib_aux = 0
#end                        [end of device unit information]
```

Figure 9-2. Syntax of UDS Input File

The following paragraphs describe the individual lines of the input file.

#version — This is a one- to four-character user version number that will be used as the new version number of the ICU. By picking consistent version numbers, you can always keep track of the latest version of your ICU.

It is important to enter meaningful data for the version number, because the ICU uses the version to determine whether the definition files are current. When the ICU is invoked by using an existing definition file, the ICU checks the version number of the definition file against the version number of the master .SCM and .TPL files. If an inconsistency occurs between these version numbers the ICU displays the differing version numbers and asks if you want to update the file. The version number that the ICU displays is built from the value you specify here, plus the date and time on which you run the ICUMRG utility. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for information on updating definition files to new versions of the ICU.

#name — The 1- to 25-character name of the driver being supported.

#abbr — The 1- to 3-character abbreviation are used to form screen names and abbreviations for all three driver screens as follows:

Screen Abbreviation	Screen Name
D_<abrv>	<name> Driver
U_<abrv>	<name> Unit Information
I_<abrv>	<name> Device-unit Information

So, if you entered an abbreviation of "ABC" and a name of "High Speed ABC," your screen abbreviations would be "D_ABC," "U_ABC," and "I_ABC." The screen names would be "High Speed ABC Driver," "High Speed ABC Unit Information," and "High Speed ABC Device-unit Information."

#driver — The value you specify here indicates the kind of driver this is and thus the kind of screens to display. The following values apply:

Value	Driver
1	Terminal Support driver with one interrupt level (see Figures 9-3, 9-8, and 9-11 for screens)
2	Terminal Support driver with two interrupt levels (see Figures 9-4, 9-8, and 9-11 for screens)
3	Interrupt-less MULTIBUS I and MULTIBUS II Full Message Passing Terminal support driver (see Figures 9-5, 9-8, and 9-11 for screens)
4	Interrupt Driven Random Access Support and common drivers (see Figures 9-6, 9-9, and 9-12 for screens)
5	MULTIBUS II Full Message Passing Random Access devices.
6	Reserved
7	General driver (see Figures 9-8, 9-11, and 9-14 for screens)

#device — This field indicates the start of the information that applies to the device information screen. This information continues until a #end field appears.

#dev_aux — This field indicates the number of auxiliary parameters on the device information screen. This value can range from 0 to 20. If this value is four or less for terminal support devices or random access devices, or 14 or less for general devices, each auxiliary parameter is displayed on a separate line, and the parameter names you specify in the #d fields are displayed there too. If more auxiliary parameters are specified, the parameters are displayed on the device information screen in rows of five parameters each. In this case, there is no room for the parameter names and if any are entered, the UDS ignores them.

When the ICU generates a system, it places the auxiliary parameters from the device information screen in the ?ICDEVC.A28 or ?ITDEV.A28 files that it creates (where ? means the character can vary), immediately after the Device Information structure. The file that is actually altered depends on whether the device is a terminal (?ITDEV.A28) or a random/common (?ICDEV>A28) device.

#d01 — Each of these fields (#d01 through #d20) identify auxiliary parameters in the device information table. Although the identifiers for these parameters are fixed (D01 through D20), if the auxiliary parameters each fit on a single line, the 1- to 41-character parameter name you specify here (as 'parameter name', surrounded by quotes) will be included on the menu to describe the auxiliary parameter.

Even if your device information table contains too many auxiliary parameters to include a parameter name for each, you must specify the #d field for a parameter if you plan to add help information for that field. In such cases, you can specify the #d field without a parameter name, as follows:

```
#d03 =
```

You can also modify the parameter names and help information for the standard parameters that normally appear on the device information screen you selected. For example, if you are setting up a random access device and you wanted to modify the parameter name and help information for the DS field (see Figure 9-5), you could include the following information in the input file:

```
#ds = 'Size of Device Local Data [0-OFFFFH]'
```

This is the description of the DS field. You can modify the other fields in the same manner.

d01 help	This is the help information for the parameters. You MUST include help information for all parameters. The UDS assumes that the help information ends when a # appears at the start of a subsequent line, or when the maximum character count is reached. The UDS displays help information when the ICU user requests help for the corresponding parameter. Help information is limited to a maximum of 1024 characters.
#end	This field designates the end of the device, unit, or device-unit information.
#unit	This field indicates the start of the information that applies to the unit information screen. This information continues until an #end field appears.
#unit_aux	This field indicates the number of auxiliary parameters on the unit information screen. This value can range from 0 to 20. If this value is 10 or less, each auxiliary parameter is displayed on a separate line with the parameter names you specify. With more than 10 auxiliary parameters, the parameters are displayed two to a row, with no room for parameter names.

When the ICU generates a system, it places the auxiliary parameters from the unit information screen in the ?ITDEV.A28 or ?ICDEV.A28 files it creates, immediately after the Unit Information structure. The file that is actually altered depends on the type of device: ?ICDEV.A28 for common and random devices, ?ITDEV.A28 for terminal devices.

#u01	Each of these fields (#u01 through #u20) identify auxiliary parameters in the unit information screen. The identifiers for these parameters are fixed (U01 through U20). If the auxiliary parameters each fit on a single line, the 1- to 41-character parameter name you specify here (as 'parameter name', surrounded by quotes) will be included on the menu to describe the auxiliary parameter. You can also use similar fields to change the parameter names and help information for any of the standard parameters of the unit information screen.
u01 help	This is the help information for the parameters. You must include help information for all parameters. The UDS assumes that the help information ends when a # appears at the start of a subsequent line. The UDS displays the help information when the ICU user requests help for the corresponding parameter. Help information is limited to a maximum of 1024 characters.
#duib	This field indicates the start of the information that applies to the device-unit screen. The device-unit information continues until a #end field is encountered.
#duib_aux	This field indicates the number of auxiliary parameters on the device-unit information screen. This value can range from 0 to 20. Currently, the UDS does not support any auxiliary parameters; therefore, set this field as follows: `#duib_aux=0`

9.1.1.2 Device Information Screens

This section lists the different Device Information Screens that the UDS can generate. When adding support for your own driver, choose the screen that matches the way your driver expects the Device Information Table to look. All of the screens in this group can also contain auxiliary parameter lines. You should set up auxiliary parameter lines if none of the Device Information Screens listed contain enough fields to support the needs of your driver. Figures 9-3 through 9-8 illustrate the screens available via the UDS utility. Figure 9-3 shows how one column of auxiliary parameter lines look if you do not add your own parameter names. Figure 9-4 shows the auxiliary parameters look when multiple columns are needed.

The meanings of the individual fields in these screens are the same as the fields in the Device Information Table. Refer to Chapters 5 or 6 for more information about those tables.

```
(D_xxx)  One-Interrupt Terminal Device Information

(DEV)    Device Name [1-16 Chars]
(NT)     Number of terminals on this controller          00
(DS)     Driver Data Size [0-0FFFFH]                     00000H
(SS)     Drivers Stack Size [0-0FFFFH]                   00000H
(INI)    Term_init Procedure Name [1-31 Chars]
(FIN)    Term_finish Procedure Name [1-31 Chars]
(SET)    Term_setup Procedure Name [1-31 Chars]
(OUT)    Term_out Procedure Name [1-31 Chars]
(ANS)    Term_answer Procedure Name [1-31 Chars]
(HAN)    Term_hangup Procedure Name [1-31 Chars]
(UTI)    Term_utility Procedure Name [1-31 Chars]
(IL)     Interrupt Level [Encoded Level]                 000H
(CHK1)   Term-check for this Level [1-31 Chars]
(D01)    Auxiliary 1                                     00000H
(D02)    Auxiliary 2                                     00000H
(D03)    Auxiliary 3                                     00000H
(D04)    Auxiliary 4                                     00000H
```

Figure 9-3. UDS Device Information Screen for One-Interrupt Terminal

```
(D_xxx)  Two-Interrupt Terminal Device Information

(DEV)    Device Name [1-16 Chars]
(NT)     Number of terminals on this controller     00
(DS)     Driver Data Size [0-0FFFFH] 00000H
(SS)     Drivers Stack Size [0-0FFFFH]  00000H
(INI)    Term_init Procedure Name [1-31 Chars]
(FIN)    Term_finish Procedure Name [1-31 Chars]
(SET)    Term_setup Procedure Name [1-31 Chars]
(OUT)    Term_out Procedure Name [1-31 Chars]
(ANS)    Term_answer Procedure Name [1-31 Chars]
(HAN)    Term_hangup Procedure Name [1-31 Chars]
(UTI)    Term_utility Procedure Name [1-31 Chars]
(IL1)    First Interrupt Level [Encoded level]      000H
(CK1)    Term-check for First Level [1 - 31 Chars]
(IL2)    Second Interrupt Level [Encoded level]     00000H
(CK2)    Term-check for Second Level [1 - 31 Chars]
(D01)    00000H  (D02)   00000H  (D03)   00000H  (D04)   00000H  (D05)   00000H
(D06)    00000H  (D07)   00000H  (D08)   00000H  (D09)   00000H  (D10)   00000H
(D11)    00000H  (D12)   00000H  (D13)   00000H  (D14)   00000H  (D15)   00000H
(D16)    00000H  (D17)   00000H  (D18)   00000H  (D19)   00000H  (D20)   00000H
```

Figure 9-4. UDS Device Information Screen for Two-Interrupt Terminal

```
(D_xxx) Interrupt-less MULTIBUS I and MULTIBUS II Full Message Passing
        Terminal Device Information

(DEV)   Device Name [1-16 Chars]
(NT)    Number of terminals on this controller          000
(DS)    Driver Data Size [0-0FFFFH]                     00000H
(SS)    Driver Stack Size [0-0FFFFH]
(INI)   Term-init Procedure Name [1-31 Chars]
(FIN)   Term-finish Procedure Name [1-31 Chars]
(SET)   Term-setup Procedure Name [1-31 Chars]
(OUT)   Term-out Procedure Name [1-31 Chars]
(ANS)   Term-answer Procedure Name [1-31 Chars]
(HAN)   Term-hangup Procedure Name [1-31 Chars]
(UTI)   Term-utility Procedure Name [1-31 Chars]
(MTP]   Message Task Priority [0-255]                   000
(CHK)   Term-check Procedure Name [1-31 Chars]
(D01)   Auxiliary 1                                     00000H
(D02)   Auxiliary 2                                     00000H
(D03)   Auxiliary 3                                     00000H
(D04)   Auxiliary 4                                     00000H
```

Figure 9-5. UDS Device Information Screen for Interruptless Terminal Devices

```
(D_xxx)  MULTIBUS I Random Access Device Information

(DEV)   Device Name [1-16 Chars]
(IL)    Interrupt Level [Encoded Level]                 000H
(ITP)   Interrupt Task Priority [0-255]                 000H
(SS)    Interrupt Procedure Stack Size [0-0FFFFH]       00000H
(DS)    Device Local Data Size [0-0FFFFH]               00000H
(NU)    Number of Units on this Device [0-255]          000H
(INI)   Initialization Procedure Name [1-31 Chars]
(FIN)   Finish Procedure Name [1-31 Chars]
(STR)   Start Procedure Name [1-31 Chars]
(STP)   Stop Procedure Name [1-31 Chars]
(INT)   Interrupt Procedure Name [1-31 Chars]
(ITO)   Interrupt Time Out [0-0FFFFH]
(D01)   Auxiliary 1                                     00000H
(D02)   Auxiliary 2                                     00000H
(D03)   Auxiliary 3                                     00000H
(D04)   Auxiliary 4                                     00000H
```

Figure 9-6. UDS Device Information Screen for Random Access Device

```
(D_xxx)

(DEV)   Device Name [1-16 Chars]
(MTP)   Message Task Priority [0-255]                      000
(SS)    Message Procedure Stack Size [0-0FFFFH]            00000H
(DS)    Device Local Data Size [0-0FFFFH]                  00000H
(NU)    Number of Units on this Device [0-255]             000
(INI)   Init Procedure Name [1-31 Chars]
(FIN)   Finish Procedure Name [1-31 Chars]
(STR)   Start Procedure Name [1-31 Chars]
(STP)   Stop Procedure Name [1-31 Chars]
(MSG)   Message Procedure Name [1-31 Chars]
(MTO)   Message Time Out [0-0FFFFH]                        00000H
(MQL)   Message Queue Length [0-0FFFFH]                    00000H
(BIN)   Board Instance [0-0FFH]                            000H
(BID)   Board ID. [1-10 Chars]
(D01)   Auxiliary 1                                        00000H
(D02)   Auxiliary 2                                        00000H
(D03)   Auxiliary 3                                        00000H
(D04)   Auxiliary 4                                        00000H
```

Figure 9-7. UDS Device Information Screen for MULTIBUS® II Message-Passing Random Access Device

```
(D_xxx)  General Device.  Device Information

(DEV)   Device Name [1-16 Chars]
(IL)    Interrupt Level [Encoded Level]                    000H
(ITP)   Interrupt Task Priority [0-255]                    000H
(D01)   Auxiliary 1                                        00000H
(D02)   Auxiliary 2                                        00000H
(D03)   Auxiliary 3                                        00000H
(D04)   Auxiliary 4                                        00000H
(D05)   Auxiliary 1                                        00000H
(D06)   Auxiliary 2                                        00000H
(D07)   Auxiliary 3                                        00000H
(D08)   Auxiliary 4                                        00000H
(D09)   Auxiliary 3                                        00000H
(D10)   Auxiliary 4                                        00000H
(D11)   Auxiliary 1                                        00000H
(D12)   Auxiliary 2                                        00000H
(D13)   Auxiliary 3                                        00000H
(D14)   Auxiliary 4                                        00000H
```

Figure 9-8. UDS Device Information Screen for General Device

9.1.1.3 Unit Information Screens

This section lists the Unit Information Screens that the UDS can generate. These screens are defined by placing information into a user input file, which the UDS reads. By choosing the appropriate driver type and adding the correct number of auxiliary lines to the driver's screens, the user can set up the ICU to handle the configuration of virtually any driver. All screens in this group can contain auxiliary parameter lines. If none of the Unit Information Screens listed contain enough fields to support your driver, set up auxiliary parameter lines. Figures 9-9, 9-10, and 9-11 illustrate the screens available via the UDS utility.

The meanings of the individual fields in these screens are the same as the fields in the Unit Information Table. Refer to Chapters 5 or 6 for more information about those tables.

```
(U_xxx)  Terminal Support Unit Information

(DEV)   Device Name [1-16 Chars]
(NAM)   Unit Info Name [1-16 Chars]
(CNF)   Connection flags [Encoded]                    000H
(TRF)   Terminal flags [Encoded]                      000H
(IBR)   Input BAUD Rate [0-0FFFFH]                    00000H
(OBR)   Output BAUD Rate [0-0FFFFH]                   00000H
(SN)    Scroll Number [0-0FFFFH]                      00000H
(U01)   Auxiliary 1                                   00000H
(U02)   Auxiliary 2                                   00000H
(U03)   Auxiliary 3                                   00000H
(U04)   Auxiliary 4                                   00000H
(U05)   Auxiliary 5                                   00000H
(U06)   Auxiliary 6                                   00000H
(U07)   Auxiliary 7                                   00000H
(U08)   Auxiliary 8                                   00000H
(U09)   Auxiliary 9                                   00000H
(U10)   Auxiliary 10                                  00000H
```

Figure 9-9. UDS Unit Information Screen for Terminal Device

```
(U_xxx)  Random Access Support Unit Information

(DEV)   Device Name [1-16 Chars]
(NAM)   Unit Info Name [1-16 Chars]
(TS)    Track Size [0-0FFFFH]                          00000H
(MR)    Maximum Retries [0-0FFFFH]                     00000H
(CS)    Cylinder Size [0-0FFFFH]                       00000H
(U01)   00000H  (U11)                                  00000H
(U02)   00000H  (U12)                                  00000H
(U03)   00000H  (U13)                                  00000H
(U04)   00000H  (U14)                                  00000H
(U05)   00000H  (U15)                                  00000H
(U06)   00000H  (U16)                                  00000H
(U07)   00000H  (U17)                                  00000H
(U08)   00000H  (U18)                                  00000H
(U09)   00000H  (U19)                                  00000H
(U10)   00000H  (U20)                                  00000H
```

Figure 9-10. UDS Unit Information Screen for Random Access Device

```
General Device Unit Information

(DEV)   Device Name [1-16 Chars]
(NAM)   Unit Info Name [1-16 Chars]
(U01)   Auxiliary 1                                    00000H
(U02)   Auxiliary 2                                    00000H
(U03)   Auxiliary 3                                    00000H
(U04)   Auxiliary 4                                    00000H
(U05)   Auxiliary 5                                    00000H
(U06)   Auxiliary 6                                    00000H
(U07)   Auxiliary 7                                    00000H
(U08)   Auxiliary 8                                    00000H
(U09)   Auxiliary 9                                    00000H
(U10)   Auxiliary 10                                   00000H
```

Figure 9-11. UDS Unit Information Screen for General Device

9.1.1.4 Device-Unit Information Screens

This section lists the Device-Unit Information Screens that the UDS generates. When adding support for your own driver, choose the screen that matches the way your driver expects the DUIB to look. None of the screens in this group currently allow auxiliary parameter lines. Figures 9-12, 9-13, and 9-14 illustrate the screens available via the UDS utility.

The meanings of the individual fields in these screens are the same as the fields in the DUIB. Refer to Chapter 4 for more information about the fields of the DUIB.

```
(I_xxx) Terminal Support Device-Unit Information

(DEV)   Device Name [1-16 Chars]
(NAM)   Device-Unit Name [1-13 Chars]
(UN)    Unit Number on this Device [0-OFFH]                000H
(UIN)   Unit Info Name [1-16 Chars]
(MB)    Max Buffers [0-OFFH]                               000H
```

Figure 9-12. UDS Device-Unit Information Screen for Terminal Device

All three terminal driver types use the same Device-Unit Information screen.

```
(I_xxx) Random Access  Device-Unit Information

(DEV)   Device Name [1-16 Chars]
(NAM)   Device-Unit Name [1-13 Chars]
(PFD)   Physical File Driver Required [Yes/No]                 No
(NFD)   Named File Driver Required [Yes/No]                    Yes
(SDD)   Single or Double Density Disks [Single/Double]         Double
(SDS)   Single or Double Sided Disks [Single/Double]           Double
(EFI)   8 or 5 Inch Disks [8/5]                                5
(SUF)   Standard or Uniform Format [Standard/Uniform]          Standard
(GRA)   Granularity [0-0FFFFFH]                                00000H
(DSZ)   Device Size [0-0FFFFFFFFH]                             00000000H
(UN)    Unit Number on this Device [0-0FFH]                    000H
(UIN)   Unit Info Name [1-16 Chars]
(RUT)   Request Update Timeout [0-0FFFFH]                      00000H
(NB)    Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]      00000H
(CUP)   Common Update [Yes/No]                                 Yes
(MB)    Max Buffers [0-0FFH]                                   000H
```

Figure 9-13. UDS Device-Unit Information Screen for Random Access Device

```
(I_xxx)  General Device-Unit Information

(DEV)   Device Name [1-16 Chars]
(NAM)   Device-Unit Name [1-13 Chars]
(FD)    File Drivers [0-0FFFFH Encoded]                        00000H
(FNC)   Functions [0-0FFH Encoded]                             000H
(FLG)   Flags [0-0FFH Encoded]                                 000H
(GRA)   Granularity [0-0FFFFFH]                                00000H
(DSZ)   Device Size [0-0FFFFFFFFH]                             00000000H
(UN)    Unit Number on this Device [0-0FFH]                    000H
(INI)   Initialize-I/O Proc Name [1-31 Chars]
(FIN)   Finish-I/O Procedure Name [1-31 Chars]
(QUE)   Queue-I/O Procedure Name [1-31 Chars]
(CAN)   Cancel-I/O Procedure Name [1-31 Chars]
(UIN)   Unit Info Name [1-16 Chars]
(RUT)   Request Update Timeout [0-0FFFFH]                      00000H
(NB)    Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]      00000H
(STP)   Service Task Priority [0-255]                          000H
(CUP)   Common Update [Yes/No]                                 Yes
(MB)    Max Buffers [0-0FFH]                                   000H
```

Figure 9-14. UDS Device-Unit Information Screen for General Device

9.1.1.5 Invoking the UDS Utility

Once you have created an input file that specifies how the screens for your device driver should appear, you are ready to invoke the UDS utility. To do this, ensure that the directory containing the UDS program also contains the UDS database file named UDS.SCM. Then invoke the utility by typing:

```
    UDS input-file TO output-file
```

where:

input-file	The name of the file that contains the information that will be used as input to the UDS utility. This is the file that was described in the section entitled "UDS Input File."
output-file	The name portion of the output files generated by UDS. UDS adds three-character extensions to this name when generating its output files. The two primary output files are output-file.SCM and output-file.TPL. You will use these output files as input to the ICUMRG utility. The other output file is output-file.LST, a listing file that shows exactly how the screens will appear when added to the ICU. Note that you should not name your UDS output files ICU286.SCM or ICU286.TPL.

For example, suppose you created an input file called NEWDRIVER.TXT and wanted the UDS utility to generate output files called SPECIAL.SCM and SPECIAL.TPL. To do this, you would enter the following command:

```
    uds newdriver.txt to special
```

Part of the output of the UDS utility are two files with extensions .SCM and .TPL (in the example, SPECIAL.SCM and SPECIAL.TPL). These files contain the definitions of the ICU screens for your driver. After running the UDS utility, you will use the ICUMRG utility to add these files to the ICU.

However, before running ICUMRG, examine the listing file (in the example, SPECIAL.LST). This file shows how the device information screen, the unit information screen, and the device-unit information screen will look when added to the ICU. If there is a problem with the appearance of any of these files, you can catch the problem early and rerun UDS, instead of adding incorrect screens to the ICU.

9.1.1.6 UDS Error Messages

If you make a mistake when creating the files to use as input to UDS, the UDS utility will generate an error message and display it on your screen. The following error messages are displayed when an external file or memory type error occurs, and are preceded by the error message:

```
***Error in UDS
```

The external file and memory type error messages are

- `*** Cannot Attach Input File`

  You did not have the proper permission to access the file containing the UDS instructions.

- `*** not enough memory for buffers*`

  Your memory partition is not large enough to permit the UDS utility to run.

- `*** Cannot Attach UDS SCM File`

  UDS needs to access a file called UDS.SCM, but you do not have read access to that file.

- `*** Invalid UDS.SCM File`

  The UDS file UDS.SCM has been corrupted.

- `***Cannot Create New SCM File`

  UDS cannot create the output file (output_file.SCM).

- `***Cannot Create New TPL File`

  UDS cannot create the output file (output_file.TPL).

- `***Cannot Create LST File`

  UDS cannot create the listing file (output_file.LST).

In addition to external file and memory type error messages, the UDS produces input file error messages, which are preceded by this message:

- `*** Error in UDS Input File on line <line-number>`

Where <line-number> is the line number in the user input file at which the error occurred. Input file error messages can be any of the following:

- `*** Missing User Version`

  There was no #version statement in your UDS input file. This statement is required.

- `*** Illegal Version`

The #version number in the user input file is outside the legal range of 1 to 4 characters.

- `*** Missing User Device Name`

  The #name field is not in the UDS input file. This statement is required.

- `*** Illegal Device Name`

  The #name identifier is of 0 length or greater than 25 characters in length.

- `*** Missing User Device Abbr`

  There was no #abbr identifier in your UDS input file. This statement is required.

- `*** Illegal Device Abbr`

  The #abbr value in the user input file is outside the legal range of 1 to 3 characters.

- *** Missing User Driver Type

  The #driver identifier is missing from the user input file. This identifier is required.

- `*** Illegal Driver Type`

  The #driver value in the user input file is outside the legal range of 1 to 7.

- `*** Missing User Device`

  The #device identifier is missing from the user input file. This identifier is required.

- `*** Missing Number of Device Auxiliaries`

  The #dev_aux identifier is missing from the user input file. This identifier is required.

- `*** Missing User Unit`

  The #unit identifier is missing from the user input file. This identifier is required.

- `*** Missing Number of Unit Auxiliaries`

  The #unit_aux identifier is missing from the user input file.

- `*** Missing User Duib`

  The #duib identifier is missing from the user input file.

- `*** Missing Number of DUIB auxiliaries`

  The #duib_aux identifier is missing from the user input file. This identifier is required.

- `*** DUIB Screen Can Not Have Auxiliary Fields`

  The #duib_aux value in the user input file is set to a value other than zero (0).

- `*** Missing Equal Sign`

  The equal sign is missing from an identifier that requires one.

- `*** Line Too Long`

A line in the user input file is longer than the allowable 132 characters.

- `*** Missing Auxiliary Help Message`

  An auxiliary parameter line was added without its required help message.

- `*** Auxiliary Line Out of Sequence`

  Auxiliary parameter lines must be listed sequentially, beginning with line 01.

- `*** Less Auxiliary Lines than Expected`

  The number of auxiliary lines is less than the value indicated in the xxx_aux value of the user input file.

- `*** More Auxiliary Lines than Expected`

  The number of auxiliary lines is more than the value indicated in the xxx_aux value of the user input file.

- `*** Illegal Input`

  Extra characters were entered on a line after the valid input.

- `*** Invalid Abbreviation`

  The abbreviation for an auxiliary field is outside the legal range of 1 to 3 characters.

- `*** Abbreviation Not Found`

  When changing a standard parameter line or its help message, the abbreviation was entered incorrectly.

- `*** Number Exceeds Maximum`

  The value of dev_aux or unit_aux is greater than 20.

- `*** Number Expected`

  A non-numeric value was entered.

- `*** Syntax Error`

  The opening quote on a parameter name line is missing.

- `*** Do Not Use # Sign in Text`

  A parameter name contains a pound symbol (#).

- `*** Do Not Use ( Sign in Text`

  A parameter name contains a left parenthesis "(".

- `*** Missing End of Text Sign`

  The closing quote on a parameter name line is missing.

- `*** Text Line Too Long`

  A parameter name exceeds 41 characters.

- `*** Help Message is too Long`

  The Help message you entered exceeds 1024 characters in length.
- `*** Field Name Expected`

  A blank line was detected in the device, unit, or duib information.
- `*** Unexpected eof`

  The user input file is incomplete.

The following error message indicates that the specified file or directory lacks read or creation permission.

- `*** I/O Error in File [file-name]`

9.1.2 ICUMRG UTILITY

After using UDS to generate .SCM and .TPL files for your new driver, invoke the ICUMRG utility to combine the information in these files with the information in the ICU286.SCM and ICU286.TPL files (the files that contain the definitions of all the other ICU screens). Before running ICUMRG, make sure that ICU286.SCM and ICU286.TPL reside in the same directory as the ICUMRG command. Then, invoke the ICUMRG utility as follows:

```
    ICUMRG input-file TO output-file
```

Where:

input-file	The name (minus the extension part) of the .SCM and .TPL files generated by the UDS. For example, if the UDS utility created files called SPECIAL.SCM and SPECIAL.TPL, you would specify the name SPECIAL here.
output-file	The name (minus the extension part) of new ICU files that ICUMRG will create. For example, if you specified the name ICUNEW, the ICUMRG utility will create files called ICUNEW.SCM and ICUNEW.TPL. These new files will contain the complete definition of the ICU, including the screens you just defined for your new driver. By naming these files something other than ICU286, you can preserve the previous version of the ICU files. However, to have the new files take effect when the ICU is run, you must change their names to ICU286.SCM and ICU286.TPL. For test purposes, you can change the name of the ICU executable file to match the base name of the new file (e.g., ICUNEW).

Once you finish adding support for your drivers to the ICU with the UDS and ICUMRG utilities, you can configure those drivers almost as you would any Intel-supplied drivers. Simply invoke the ICU and go to the "(UDDM) UDS Device Drivers Module." Enter the appropriate driver type (T)erminal or (C)ommon and the full pathname for the location of the object code for your device driver. After you enter the correct value, choose the device you want to configure. Then fill in the appropriate values when the ICU displays the Device Information, Unit Information, and Device-Unit Information screens.

9.1.2.1 UDS Modules Screen in the ICU

```
  (UDDM)          UDS   Device Driver Modules
         Module= Driver type , Object code pathname
                     [T/C]    ,  [1-55 Characters]
  [ 1]   Module=
```

You specify (C) for common/random/custom Object Module pathname(s) and (T) for terminal drivers.

All drivers added via the UDS must be placed in modules according to type. All of your terminal modules must be located in one module, all of your common/random/custom drivers must be located in a separate module. For example, 1 = T, TERMINAL.LIB and 2 = C, DRIVER.LIB

NOTE

Before changing the name of any ICUMRG output files to ICU286.SCM and ICU286.TPL, save the original files by copying them to other files (such as ICU286OLD.SCM and ICU286OLD.TPL). Although ICUMRG allows you to add support for new drivers, once you add that support, there is no way to remove it. If you decide you don't want the ICU to display information about one of your drivers, or you made a mistake when you added the driver support, you must revert back to the original ICU286.SCM and ICU286.TPL files (or an intermediate version that didn't contain support for that driver).

9.2 ADDING YOUR DRIVER AS A CUSTOM DRIVER

If you don't want to modify the ICU, you can add your driver as a custom device driver. Perform the following steps to do this:

1. Ascertain the device numbers and device-unit numbers to use in the DUIBs for your devices, as follows:

a. Use the ICU to configure a system containing all the Intel-supplied and ICU-supported user drivers you require.

b. Use the G option to generate that system.

c. Use a text editor to examine the file ?ICDEV.A28 (the ? means the first letter can vary). Among other things, this file contains DUIBs for all the device-units you defined in your configuration.

d. Look for the %DEVICETABLES macro that appears after all the DEFINE_DUIB structures. The second and third parameters in that macro list are the next available device-unit number and device number, respectively. For example, suppose the %DEVICETABLES macro appears as follows:

```
%DEVICETABLES(NUMDUIB,0000CH,005H,003E8H)
```

In this case, the next available device-unit number is 0CH and the next available device number is 05H.

e. Use the next available device number and device-unit number in your DUIBs.

2. Create the following:

a. A file containing the DUIBs for all the device-units you are adding. Use the DEFINE_DUIB structures shown in Chapter 4. Place all the structures in the same file. Later, the ICU includes this file in the assembly of the ?ICDEV.A28 file.

b. A file containing all the device information tables of the random/common/custom type that you are adding. Use the RADEV_DEV_INFO structures shown in Chapter 4 for any random access drivers you add. Use a structure similar to the one shown in Chapter 6 for terminal drivers. Later, the ICU includes this file in the assembly of the ?ICDEV.A28 file.

c. If applicable, any random access or common unit information table(s). Use the RADEV_UNIT_INFO structures shown in Chapter 4 for any random access drivers you add. Add these tables to the file created in step b.

d. A file containing all the device information tables of the terminal type that you are adding. Use the RADEV_DEV_INFO structures shown in Chapter 4 for any random access drivers you add. Use a structure similar to the one shown in Chapter 6 for terminal drivers. Later, the ICU includes this file in the assembly of the ?ITDEV.A28 file.

e. If applicable, any terminal unit information table(s). Use a structure similar to the one shown in Chapter 6 for terminal drivers. Add these tables to the file created in step b.

f. External declarations for any procedures that you write. The names of these procedures appear in either the DUIB or the Device Information Table associated with this device driver. Add these declarations to the file created in step b and d.

3. Use the ICU to configure your final system. When doing so:

   a. Answer "yes" when asked if you have any device drivers not supported by the ICU.

   b. As input to the "Custom User Devices" screen, enter the pathname of your random/common/custom device driver library. This refers to the library built earlier; for example, :F1:DRIVER.LIB.

   c. As input to the "Custom User Devices" screen, enter the pathname of your terminal device driver library. This refers to the library built earlier; for example, :F1:TERMINAL.LIB.

   d. Also, enter the information the ICU needs to include your configuration data in the assembly of ?ICDEV.A28 and ?ITDEV.A28. The information needed includes the following:

      - DUIB source code pathname (the file created in step 2a).
      - Device and Unit source code pathnames (the files created in steps 2b through 2f).
      - Number of user defined devices.
      - Number of user defined device-units.

The ICU does the rest.

Figure 9-15 contains an example of the "Custom User Devices" screen. The bold text represents user input to the ICU. In this example, the file :F1:DRIVER.LIB contains the object code for the random/common/custom drivers; the file :F1:TERMINAL.LIB contains the object code for the terminal driver. F1:DUIB contains the source code for the DUIBs, and :F1:RINFO.INC contains the source code for the Device and Unit Information Tables along with the necessary external procedure declarations for the random/common/custom drivers. TINFO.INC contains the source code for the Device and Unit Information Tables and the necessary external procedure declarations for the terminal driver.

The code in the DRIVER.LIB file supports one device with two units. The code in TERMINAL.LIB supports one device with 2 units; therefore, the (ND) Number of User Defined Devices [0-0FFH] field equals 2, the (NDU) Number of User Defined Device-Units [0-0FFH] field equals 4. Refer to the Extended iRMX II Interactive Configuration Utility Reference Manual for instructions on how to use the ICU.

```
 (USERD)     User Devices
 (OPN)   Random Access Object Code Path Name [1-45 Chars/NONE]
                                                              NONE
 (TOP)   Terminal Object Code Path Name [1-45 Chars/NONE]
                                                              NONE
 (OPN)   Duib Source Code Path Name [1-45 Chars/NONE]
                                                              NONE
 (DUP)   Random Access Device and Unit Source Code Path Name [1-4 Chars/NONE]
                                                              NONE
 (TUP)   Terminal Device and Unit Source Code Path Name [1-45 Chars/NONE]
                                                              NONE
 (ND)    Number of User Defined Devices [0-0FFH]              0H
 (NDU)   Number of User Defined Device-Units [0-0FFH]         0H

     (N01) NONE     (N02) NONE     (N03) NONE
     (N04) NONE     (N05) NONE     (N06) NONE
     (N07) NONE     (N08) NONE     (N09) NONE
     (N10) NONE     (N11) NONE     (N12) NONE
     (N13) NONE     (N14) NONE     (N15) NONE
     (N16) NONE     (N17) NONE     (N18) NONE

: OPN = :F1:DRIVER.LIB <CR>
: TOP = :F1:TERMINAL.LIB <CR>
: OPN = :F1:DUIB.INC <CR>
: DUP = :F1:RINFO.INC <CR>
: TUP = :F1:TINFO.INC <CR>
: ND = 2 <CR>
: NDU = 4 <CR>
```

Figure 9-15. Example User Devices Screen

9.2.1 Example of Adding an Existing Driver as a Custom Driver

This section illustrates how to create the screens needed for adding the iSBC 544A device to you system using the UDS. Because the configuration of devices in an iRMX system is complex, this example covers the process in detail.

While reading this example keep in mind that the code for terminal drivers is in a different segment than the code for random or common drives. Because of this split in the segments, care must be taken to properly provide the correct PUBLICS, EXTRNS, and NOPUBLICS EXCEPT, and also to properly bind the code segments together.

```
(USERD)         User Devices
(OPN) Random Access Object Code Path Name [1-45 Chars/NONE]
                                                                NONE
(TOP) Terminal Object Code Path Name [1-45 Chars/NONE]
                                                                NONE
(DPN) Duib Source Code Path Name [1-45 Chars/NONE]
                                                                DUIB.INC
(DUP) Random Access Device and Unit Source Code Path Name [1-45 Chars/NONE]
                                                                NONE
(TUP) Terminal Device and Unit Source Code Path Name [1-45 Chars/NONE]
                                                                TINFO.INC
(ND)  Number of User Defined Devices [0-0FFH]               01H
(NDU) Number of User Defined Device-Units [0-0FFH]          04H
     Terminal Device and Unit Information Names [1-16 Chars]

     (N01) DINFO_544A      (N02) UINFO_544A     (N03) NONE
     (N04) NONE      (N05) NONE      (N06) NONE
     (N07) NONE      (N08) NONE      (N09) NONE
     (N10) NONE      (N11) NONE      (N12) NONE
     (N13) NONE      (N14) NONE      (N15) NONE
     (N16) NONE      (N17) NONE      (N18) NONE
```

TOP was left at NONE in this example because the iSBC 544A driver code is already in the driver library XCMDRV.LIB. If you were adding another module, you would enter the location of the file as a full path name.

OPN and DUP were left at NONE because the driver being configured is a terminal driver, not a random access, common, or custom driver.

You can add up to 18 total Terminal DINFO and UINFO public names in this screen.

9.2.1.1 Contents of the DUIB.INC file specified in the (DPN) Parameter

This section shows the contents of the file whose pathname you supplied in the (DPN) DUIB Source Code Pathname parameter of the User Devices Screen. This assembly-language file provides the information needed to define how the Operating System should interface with this device.

Note the lines with arrows pointing to them. These are the device number and device-unit number for this device. These numbers were taken from the ?ICDEV.A28 file as follows:

1. Ensure that the files you start with contain all of the Intel-supplied and ICU-supported drivers that you require. If you haven't generated such a system use the ICU to do so before continuing.

2. Use a text editor to examine the file ?ICDEV.A28 (the ? means that the first letter can vary). You will find all of the DUIBs for your entire system in this file. Scan this file for a line that starts with %DEVICETABLE >

3. %DEVICETABLE is a macro that appears below all of the systems' DEFINE_DUIB structures. The second and third parameters in that macro are the next available device-unit and device number, respectively. For example, suppose the %DEVICETABLE macro appears as follows:

```
%DEVICETABLE (NUMDUIB, 0002EH, 008H, 003E8H)
```

In this case, the next available device-unit number is 2EH and the next available device number is 08H.

4. Use these numbers to fill in the two lines of the file indicated by the arrows.

At the end of this file are several more lines that should be noted. Be sure to examine the last part of this figure and read the text that goes with it.

```
DEFINE_DUIB <
& 'T2',
& 00001H,
& 0FBH,
& 00,
& 00,
& 00,
& 00,
& 08H, <======================= Put next available DEVICE NUMBER here
& 0H,
& 2EH, <======================= Put next available DEVICE-UNIT NUMBER here
& TSINITIO,
& TSFINISHIO,
& TSQUEUEIO,
& TSCANCELIO,
& DINFO_544A,
& UINFO_544A,
& 0FFFFH,
& 0,
& 130,
& FALSE,
& 0H,
& 0
&>
DEFINE_DUIB <
& 'T3',
& 00001H,
& 0FBH,
& 00,
& 00,
& 00,
& 00,
& 08H, <======================= The DEVICE NUMBER is the same
& 0H,
& 2FH, <======================= The DEVICE-UNIT number (T3) is equal to the
& TSINITIO,                    DEVICE-UNIT number of 'T2' plus one.
& TSFINISHIO,
& TSQUEUEIO,
& TSCANCELIO,
& DINFO_544A,
& UINFO_544A,
& 0FFFFH,
& 0,
& 130,
& FALSE,
& 0H,
& 0
&>
```

Figure 9-16. Computing Device and Device-Unit Numbers and BND286 Information (Continued)

```
DEFINE_DUIB <
& 'T4',
& 00001H,
& 0FBH,
& 00,
& 00,
& 00,
& 00,
& 08H, <=========================== The DEVICE NUMBER is the same
& 0H,
& 30H, <=========================== The DEVICE-UNIT number (T4) is equal to the
& TSINITIO,                        DEVICE-UNIT number of 'T3' plus one.
& TSFINISHIO,
& TSQUEUEIO,
& TSCANCELIO,
& DINFO_544A,
& UINFO_544A,
& 0FFFFH,
& 0,
& 130,
& FALSE,
& 0H,
& 0
&>
DEFINE_DUIB <
& 'T5',
& 00001H,
& 0FBH,
& 00,
& 00,
& 00,
& 00,
& 08H, <=========================== The DEVICE NUMBER is the same
& 0H,
& 31H, <=========================== The DEVICE-UNIT number (T5) is equal to the
& TSINITIO,                        DEVICE-UNIT number of 'T4' plus one.
& TSFINISHIO,
& TSQUEUEIO,
& TSCANCELIO,
& DINFO_544A,
& UINFO_544A,
& 0FFFFH,
& 0,
& 130,
& FALSE,
& 0H,
& 0
&>
```

Figure 9-16. Computing Device and Device-Unit Numbers and BND286 Information (continued)

```
BIOS_CODE ENDS  <═══════════════════╗
TSC_CODE SEGMENT ER PUBLIC          ║
    extrn DINFO_544A : far          ╠════ New portion of file
    extrn UINFO_544A : far          ║     to account for new segment.
TSC_CODE ENDS                       ║
BIOS_CODE SEGMENT<══════════════════╝
```

Figure 9-16. Computing Device and Device-Unit Numbers and BND286 Information

The lines starting with "BIOS_CODE ENDS" through "BIOS_CODE SEGMENT" must be added to the end of the file. They provide BND286 with information on the location of your information tables. You must provide an "cxtrn <MODULE_NAME>: far' declaration for each DINFO and UINFO public name specified here, these names must be supplied as parameters N01 through N18 above. This declaration is required because all terminal information is stored in a different physical segment than other driver information, and a far call is required to access it.

9.2.1.2 Contents of the File Specified in the (TUP) Parameter

This section shows the contents of the file whose pathname you supplied in the (TUP) Terminal Device and Unit Source Code Path Name parameter of the User Devices Screen. This assembly-language file provides the information needed to define how the Operating System should interface with this device.

```
   extrn I544INIT : near
   extrn I544FINISH : near
   extrn I544SETUP : near
   extrn I544CHECK : near
   extrn I544ANSWER : near
   extrn I544HANGUP : near
   extrn I544UTILITY : near
;
          PUBLIC DINFO_544A <
   DINFO_544A DW 04H
   DW 9
   DW 300
   DW I544INIT
   DW I544FINISH
   DW I544SETUP
   DW TERMNULL
   DW I544ANSWER                                  Public Declarations
   DW I544HANGUP
   DW I544UTILITY
   DW 1
   DW 071H
   DW I544CHECK
   DD 0FE0000H
   DW 04000H
   DB 01H
             PUBLIC UINFO_544A <
   UINFO_544A  DW 01AH
   DW 0109H
   DW 02580H
   DW 00000H
   DW 012H
```

Figure 9-17. Public Declarations Needed for the DINFO and UINFO Tables

Provide the normal "extrn <MODULE_NAME>: near" declarations for term$init, ..., term$finish procedures. You must also provide a "PUBLIC <table name>" label prior to each DINFO and UINFO table specified.

9.2.1.3 Portion of System Generation SUBMIT File as Changed by this Process

After completing the changes outlined above you must generate a new system using the ICU. During the process of the system generation, information is sent to the screen. This section presents those portions of the system generation that are changed by the steps outlined above.

```
;
;     BIOS
;
   .
   .
   .
:LANG:ASM286 ICDEV.A28   <══╤═══════════ IDEVC.FILE from earlier releases
:LANG:ASM286 ITDEV.A28   <══╛            is now two files
   .
   .
   .
:LANG:bnd286 &       <═══════════════════ SEPARATE BIND OF TSC CODE SEGMENT
ITDEV.OBJ, &
/RMX286/IOS/XCMDRV.LIB(XTSIF), &
/RMX286/IOS/XCMDRV.LIB(XTSIO), &
/RMX286/IOS/XCMDRV.LIB, &
/RMX286/IOS/XDRVUT.LIB, &
:LANG:PLM286.LIB, &
/RMX286/LIB/RMXIFC.LIB &
RENAMESEG(CODE TO TSC_CODE, DATA TO TSC_DATA) &
OBJECT (TSC.LNK)  NODEBUG NOTYPE SEGSIZE(STACK(0))   &
 NOLOAD NOPUBLICS EXCEPT( TSCINITIO, &
  TSCFINISHIO, &
  DINFO_02H, &
  UINFO_8251, &
  DINFO_03H, &
  UINFO_18848, &
  DINFO_04H, &
  UINFO_546, &
  UINFO_546CC, &
  DINFO_05H, &
  UINFO_547A, &
  DINFO_06H, &
  UINFO_547B, &
  DINFO_07H, &
  UINFO_547C, &
  DINFO_544A, &   <═════════════════════ USER SPECIFIED PUBLIC DINFO
  UINFO_544A, &   <═════════════════════ USER SPECIFIED PUBLIC UINFO
  TSCQUEUEIO, &
  TSCCANCELIO)
:LANG:bnd286 &
IOS1.LNK, &
TSC.LNK, &     <═════════════════════ INCLUSION OF TSC SUBSYSTEM IN IOS
                                      SYSTEM BIND
```

Figure 9-18. Portion of the Modified SUBMIT File (continued)

```
ICDEV.OBJ, &
/RMX286/IOS/XCMDRV.LIB, &
/RMX286/IOS/XDRVUT.LIB, &
:LANG:PLM286.LIB, &
/RMX286/LIB/RMXIFC.LIB &
RENAMESEG(TSC_DATA TO DATA) &
OBJECT (IOS.LNK)  NODEBUG NOTYPE SEGSIZE(STACK(0))    &
 NOLOAD NOPUBLICS EXCEPT (rqaiosinittask , &
  ReqAttachDevice   , &
   .
   .
   .
```

Figure 9-18. Portion of the Modified SUBMIT File

APPENDIX A RANDOM ACCESS SUPPORT FOR INTERRUPT-DRIVEN DEVICES

In a MULTIBUS I system interrupt-driven devices can be either devices attached directly to the CPU board or separate controllers. In MULTIBUS II systems these devices must be attached directly to the CPU board. Interrupt-driven devices signal the CPU host via interrupts at a specified interrupt level.

This appendix describes, in general terms, the operations of the random access support routines as they apply to interrupt-driven devices. The routines described include

INIT$IO
FINISH$IO
QUEUE$IO
CANCEL$IO
INTERRUPT$TASK

These routines are supplied with the I/O System and are the device driver routines actually called when an application task makes an I/O request to support a random access or common device. These routines ultimately call the device-specific device initialize, device finish, device start, device stop, and device interrupt procedures.

This appendix provides descriptions of these routines to show you the steps that an actual device driver follows. You can use this appendix to get a better understanding of the I/O System-supplied portion of a device driver to make writing the device-dependent portion easier. Or you can use it as a guideline for writing custom device drivers.

A.1 INIT$IO PROCEDURE

The I/O System calls INIT$IO when an application task makes an RQ$A$PHYSICAL$ATTACH$DEVICE system call and no units of the device are currently attached.

INIT$IO initializes objects used by the remainder of the driver routines, creates an interrupt task, and calls a user-supplied procedure to initialize the device itself.

When the I/O System calls INIT$IO, it passes the following parameters:

- A pointer to the DUIB of the device-unit to initialize
- A pointer to the location where INIT$IO must return a token for a data segment (data storage area) that it creates
- A pointer to the location where INIT$IO must return the condition code

The following paragraphs show the general steps the INIT$IO procedure follows to initialize the device. Figure A-1 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

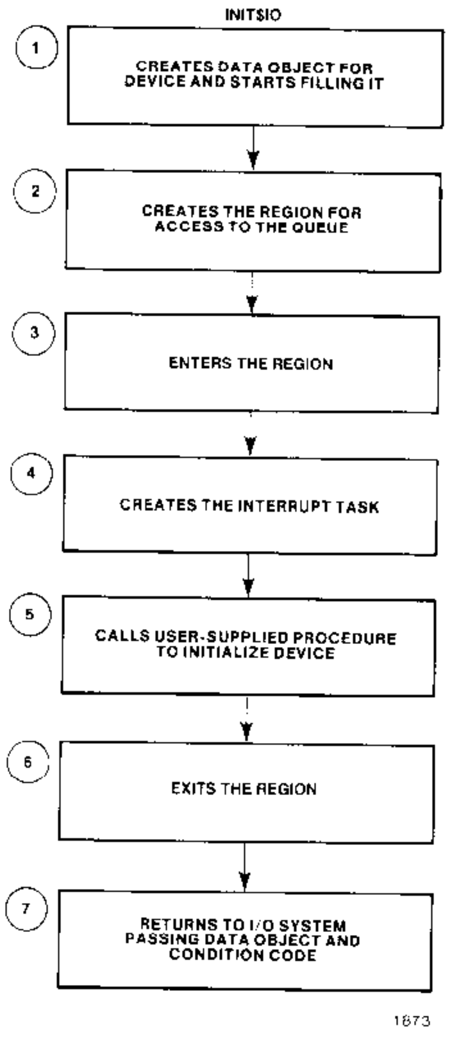

INIT$IO
1
CREATES DATA OBJECT FOR DEVICE AND STARTS FILLING IT
2
CREATES THE REGION FOR ACCESS TO THE QUEUE
3
ENTERS THE REGION
4
CREATES THE INTERRUPT TASK
5
CALLS USER-SUPPLIED PROCEDURE TO INITIALIZE DEVICE
6
EXITS THE REGION
7
RETURNS TO I/O SYSTEM PASSING DATA OBJECT AND CONDITION CODE
1873

Figure A-1. Random Access Device Driver Initialize I/O Procedure

1. It creates a data storage area to be used by all the procedures in the device driver. The size of this area depends in part on the number of units in the device and any special space requirements of the device. INIT$IO then begins initializing this area and eventually places the following information there:

   - A token for a region. Step 2 creates this region for mutual exclusion.

   - An array to contain the addresses of the DUIBs for the device-units attached to this device. INIT$IO places the address of the DUIB for the first attaching device unit into this array.

   - A token for the interrupt task.

   - Other values indicating the queue is empty and the driver is not busy.

   It also reserves space in the data storage area for device data.

2. It creates a region. The other random access support routines receive control of this region whenever they place a request on the queue or remove a request from the queue. INIT$IO places the token for this region in the data storage area.

3. It enters the region to prevent the interrupt task from starting before initialization is complete.

4. It creates an interrupt task to handle interrupts generated by this device. When INIT$IO invokes CREATE$TASK to create the interrupt task, it does not specify the task's data segment. Instead, it uses the data$seg parameter of CREATE$TASK to pass the interrupt task a token for the data storage area. This area is where the interrupt task will get information about the device. INIT$IO places the actual data segment value, as well as a token for the interrupt task, in the data storage area.

5. It calls a device-specific device initialization procedure that initializes the device itself. It gets the address of this procedure by examining the Device Information Table specified in the DUIB. Refer to Chapter 5 for information on how to write this initialization procedure.

6. It exits the region.

7. It returns control to the I/O System, passing a token for the data storage area and a condition code which indicates the success of the initialization operation.

If an error occurs at any point in these steps, the INIT$IO procedure exits the region, deletes all the objects it has created up to that point, and returns an error to the I/O System.

A.2 FINISH$IO PROCEDURE

The I/O System calls FINISH$IO when an application task makes an RQ$A$PHYSICAL$DETACH$DEVICE system call and no other units of the device are currently attached.

FINISH$IO calls a user-supplied procedure to perform final processing on the device itself, deletes the interrupt task, and deletes objects used by the other device driver routines.

When the I/O System calls FINISH$IO, it passes the following parameters:

- A pointer to the DUIB of the device-unit just detached
- A TOKEN for the data storage area created by INIT$IO

The following paragraphs show the general steps that the FINISH$IO procedure goes through to terminate processing for a device. Figure A-2 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It calls a device-specific device finish procedure that performs any necessary final processing on the device itself. FINISH$IO gets the address of this procedure by examining the Device Information Table specified in the DUIB. Refer to the Chapter 5 for information about device information tables.
2. It deletes the interrupt task originally created for the device by the INIT$IO procedure and cancels the assignment of the interrupt handler to the specified interrupt level.
3. It deletes the region and the data storage area originally created by the INIT$IO procedure, allowing the operating system to reallocate the memory used by these objects.
4. It returns control to the I/O System.

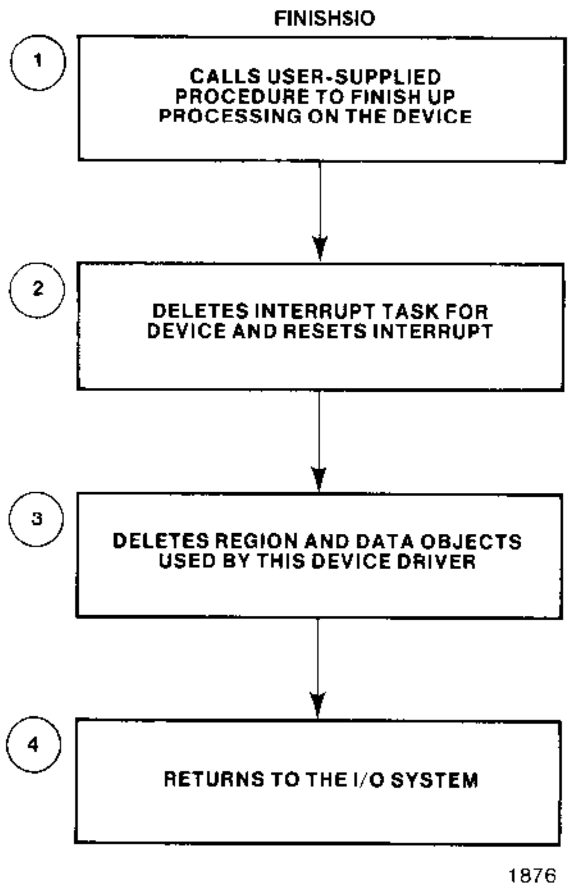

FINISHSIO
1
CALLS USER-SUPPLIED PROCEDURE TO FINISH UP PROCESSING ON THE DEVICE
2
DELETES INTERRUPT TASK FOR DEVICE AND RESETS INTERRUPT
3
DELETES REGION AND DATA OBJECTS USED BY THIS DEVICE DRIVER
4
RETURNS TO THE I/O SYSTEM
1876

Figure A-2. Random Access Device Driver Finish I/O Procedure

A.3 QUEUE$IO PROCEDURE

The I/O System calls the QUEUE$IO procedure to place an I/O request on a queue of requests. This queue has the structure of the doubly-linked list shown in Figure 7-1. If the device itself is not busy, QUEUE$IO also starts the request.

When the I/O System calls QUEUE$IO, it passes the following parameters:

- A token for the IORS
- A pointer to the DUIB
- A token for the data storage area originally created by INIT$IO

The following paragraphs show the general steps that the QUEUE$IO procedure goes through to place a request on the I/O queue. Figure A-3 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It sets the DONE field in the IORS to 0H, indicating the request has not yet been completely processed. Other procedures that start the I/O transfers and handle interrupt processing also examine and set this field. It also sets IORS.STATUS to E$OK and IORS.ACTUAL to 0H.
2. It receives control of the region and thus access to the queue. This allows QUEUE$IO to adjust the queue without concern that other tasks might also be doing this at the same time.
3. It verifies that the request is within the range of zero to device size for this device. If the request is outside this range, QUEUE$IO returns E$PARAM. For a valid request, it converts IORS.DEV$LOC from the absolute byte position on the device, as passed by the BIOS, to the absolute block (sector) number (if track size equals zero). If the track size is not zero IORS.DEV$LOC is converted to the sector and track number. Finally, it places the IORS on the queue in seek-optimized order.
4. If the device is busy processing an I/O request, QUEUE$IO goes on to Step 5. Otherwise, it calls the device-specific device start procedure to process the request at the head of the queue. This start procedure is described in Chapter 5.
5. It surrenders control of the region, thus allowing other routines to have access to the queue.

NOTE

If the request is complete, QUEUE$IO returns the IORS to the response mailbox; if not, the interrupt task returns it upon completion. The random access support does not return a CLOSE request until all prior requests for the same unit are completed.

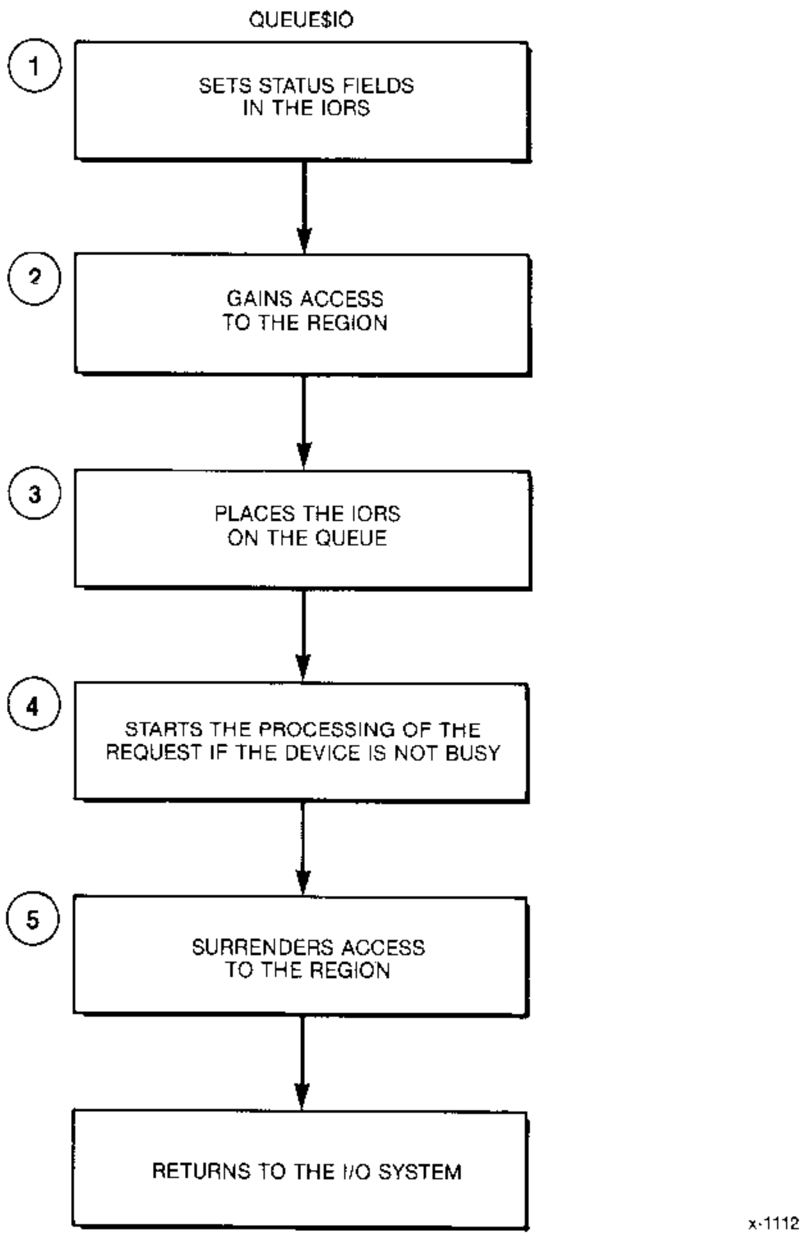

QUEUE$IO
1
SETS STATUS FIELDS IN THE IORS
2
GAINS ACCESS TO THE REGION
3
PLACES THE IORS ON THE QUEUE
4
STARTS THE PROCESSING OF THE REQUEST IF THE DEVICE IS NOT BUSY
5
SURRENDERS ACCESS TO THE REGION
RETURNS TO THE I/O SYSTEM
x-1112

Figure A-3. Random Access Device Driver Queue I/O Procedure

A.4 CANCEL$IO PROCEDURE

The I/O System calls CANCEL$IO to remove one or more requests from the queue and possibly to stop the processing of a request, if it has already been started. The I/O System calls this procedure in one of two instances:

- If a task invokes the RQ$A$PHYSICAL$DETACH$DEVICE system call and specifies the hard detach option (refer to the *Extended iRMX II Basic I/O System Calls* manual for information about this system call). The hard detach removes all requests from the queue.
- If the job containing the task that makes an I/O request is deleted. In this case, the I/O System calls CANCEL$IO to remove all of that task's requests from the queue.

When the I/O System calls CANCEL$IO, it passes the following parameters:

- An ID value that identifies requests to be canceled
- A pointer to the DUIB
- A token for the device data storage area

The following paragraphs show the general steps that the CANCEL$IO procedure goes through to cancel an I/O request. Figure A-4 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It receives access to the queue by gaining control of the region. This allows it to remove requests from the queue without concern that other tasks might also be processing the IORS at the same time.
2. It locates the request(s) to be canceled by looking at the cancel$id field of the queued IORSs, starting at the front of the queue.
3. If the request that is to be canceled is at the head of the queue, that is, the device is processing the request, CANCEL$IO calls a device-specific device stop procedure that stops the device from further processing. Refer to the Chapter 5 for information on how to write this device stop procedure.
4. If the request is finished, or if the IORS is not at the head of the queue, CANCEL$IO removes the IORS from the queue and sends it to the response mailbox indicated in the IORS. It examines the rest of the requests on the queue, removing all of them whose cancel$id fields match the ID of the canceled request.
5. It surrenders control of the region, thus allowing other procedures to gain access to the queue.

NOTE

The additional CLOSE request supplied by the I/O System will not be processed until all other requests with the given cancel$id value have been dealt with.

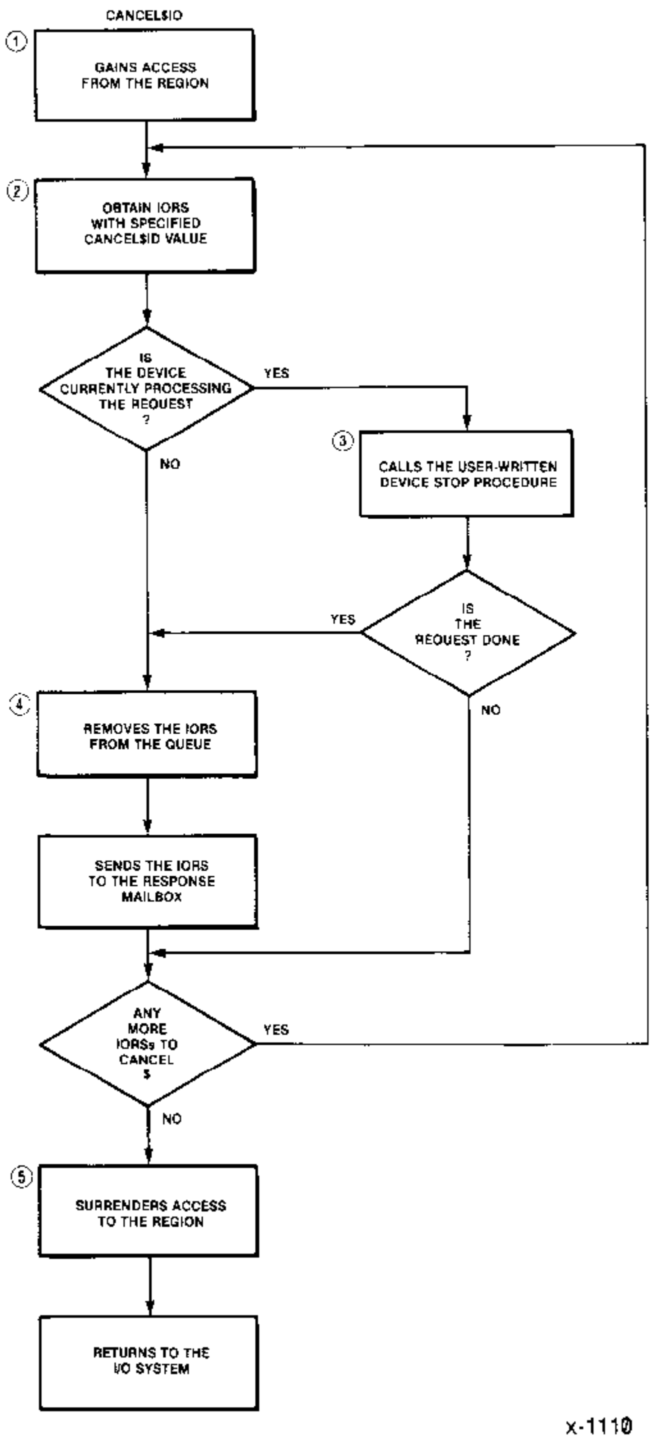

CANCEL$IO
1
GAINS ACCESS FROM THE REGION
2
OBTAIN IORS WITH SPECIFIED CANCEL$ID VALUE
IS THE DEVICE CURRENTLY PROCESSING THE REQUEST ?
YES
NO
3
CALLS THE USER-WRITTEN DEVICE STOP PROCEDURE
IS THE REQUEST DONE ?
YES
NO
4
REMOVES THE IORS FROM THE QUEUE
SENDS THE IORS TO THE RESPONSE MAILBOX
ANY MORE IORSs TO CANCEL ?
YES
NO
5
SURRENDERS ACCESS TO THE REGION
RETURNS TO THE I/O SYSTEM
x-1110

Figure A-4. Random Access Device Driver Cancel I/O Procedure

A.5 INTERRUPT TASK (INTERRUPT$TASK)

As a part of its processing, the INIT$IO procedure creates an interrupt task for the entire device. This interrupt task responds to all interrupts generated by the units of the device, processes those interrupts, and starts the device working on the next I/O request on the queue.

The following paragraphs show the general steps that the interrupt task for the random access device driver goes through to process a device interrupt. Figure A-5 illustrates these steps. The numbers in Figure A-5 correspond to the step numbers in the text.

1. It uses the contents of the processor's DS register to obtain a token (identifier) for the device data storage area. This is possible because of the following two reasons:

    - When INIT$IO created the interrupt task, instead of specifying the interrupt task's DS register in the data$seg parameter of the CREATE$TASK call, it passed the token of the data storage area in this parameter. Therefore, when the Nucleus created the task, it set the task's DS register to the value of the token.

    - When the INIT$IO procedure initialized the data storage area, it included the value of the interrupt task's DS register there.

    When the interrupt task starts running, it saves the contents of the DS register (to use as the address of the data storage area) and sets the DS register to the value listed in the data storage area. Thus the DS register does point to the task's data segment, and the task also knows the address of the data storage area. This is the mechanism that is used to pass the address of the device's data storage area from the INIT$IO procedure to the interrupt task.

2. It invokes the RQ$SET$INTERRUPT system call to indicate that it is an interrupt task associated with the interrupt handler supplied with the random access device driver. It also indicates the interrupt level to which it will respond (it obtains this information from the Device Information Table).

3. It begins an infinite loop by invoking the RQE$TIMED$INTERRUPT system call to wait for an interrupt of the specified level. If the time limit expires before an interrupt occurs, the effect is the same as a null (or spurious) interrupt, and the task waits for another interrupt. By invoking a number of RQE$TIMED$INTERRUPT calls, instead of a single WAIT$INTERRUPT, the task allows lower-priority tasks to gain control between calls. For example, if an application attempts to send data to a line printer that isn't connected, the user can press CONTROL-C to cancel the operation.

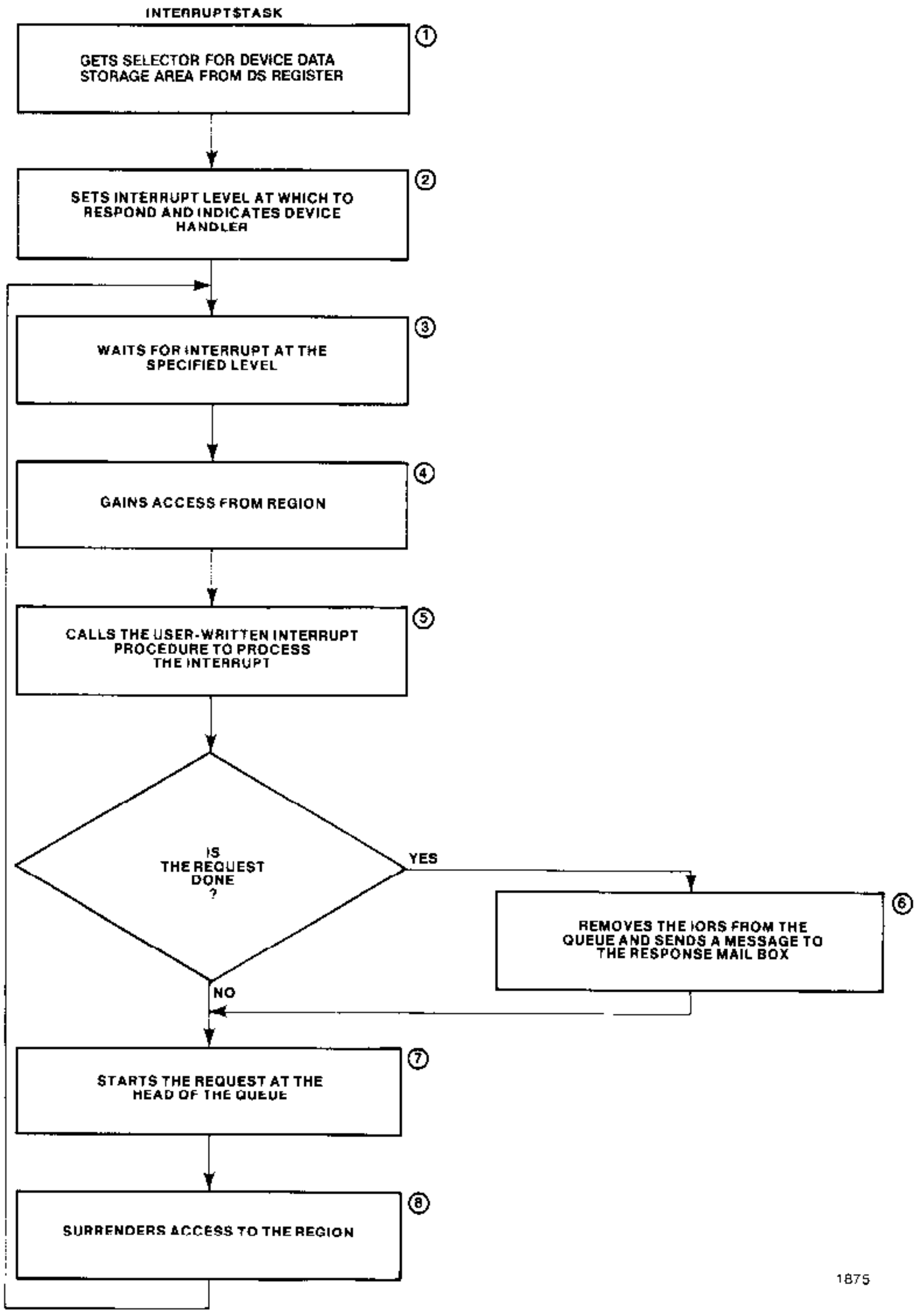

INTERRUPT$TASK
① GETS SELECTOR FOR DEVICE DATA STORAGE AREA FROM DS REGISTER
② SETS INTERRUPT LEVEL AT WHICH TO RESPOND AND INDICATES DEVICE HANDLER
③ WAITS FOR INTERRUPT AT THE SPECIFIED LEVEL
④ GAINS ACCESS FROM REGION
⑤ CALLS THE USER-WRITTEN INTERRUPT PROCEDURE TO PROCESS THE INTERRUPT
IS THE REQUEST DONE ?
YES
NO
⑥ REMOVES THE IORS FROM THE QUEUE AND SENDS A MESSAGE TO THE RESPONSE MAIL BOX
⑦ STARTS THE REQUEST AT THE HEAD OF THE QUEUE
⑧ SURRENDERS ACCESS TO THE REGION
1875

Figure A-5. Random Access Device Driver Interrupt Task

4. Via a region, it gains access to the request queue. This allows it to examine the first entry in the request queue without concern that other tasks are modifying it at the same time.
5. It calls a device-specific device-interrupt procedure to process the actual interrupt. This can involve verifying that the interrupt was legitimate or any other operation that the device requires. This interrupt procedure is described further in Chapter 5.
6. If the request has been completely processed, (one request can require multiple reads or writes, for example), the interrupt task removes the IORS from the queue and sends it as a message to the response mailbox (exchange) indicated in the IORS. If the request is not completely processed, the interrupt task leaves the IORS at the head of the queue.
7. If there are requests on the queue, the interrupt task initiates the processing of the next I/O request by calling the device-specific device-start procedure.
8. In any case, the interrupt task then surrenders access to the queue, allowing other routines to modify the queue, and loops back to wait for another interrupt.

APPENDIX B RANDOM ACCESS SUPPORT FOR MESSAGE-PASSING DEVICES

In an interrupt-driven system, the CPU host and the controller communicate via hardware interrupts. In systems having full message-passing capabilities, this communication is done via virtual interrupts (more precisely noted as messages throughout this appendix). For an overview and examples of the message-passing process, see the *Extended iRMX II Nucleus User's Guide.*

This appendix describes, in general terms, the operations of the random access support routines as they apply to message-passing devices. The routines described include

INIT$IO
FINISH$IO
QUEUE$IO
CANCEL$IO
MESSAGE$TASK

These routines are supplied with the I/O System and are the device driver routines actually called when an application task makes an I/O request to support a random access or common device. These routines ultimately call the device-specific device initialize, device finish, device start, device stop, and device interrupt procedures.

This appendix provides descriptions of these routines to show you the steps that an actual device driver follows. You can use this appendix to get a better understanding of the I/O System-supplied portion of a device driver to make writing the device-dependent portion easier. Or you can use it as a guideline for writing custom device drivers.

B.1 INIT$IO PROCEDURE

The I/O System calls INIT$IO when an application task makes an RQ$A$PHYSICAL$ATTACH$DEVICE system call and no units of the device are currently attached.

INIT$IO initializes objects used by the remainder of the driver routines, creates a message task, and calls a user-supplied procedure to initialize the device itself.

When the I/O System calls INIT$IO, it passes the following parameters:

- A pointer to the DUIB of the device-unit to initialize

- A pointer to the location where INIT$IO must return a token for a data segment (data storage area) that it creates
- A pointer to the location where INIT$IO must return the condition code

The following paragraphs show the general steps the INIT$IO procedure follows to initialize the device. Figure B-1 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

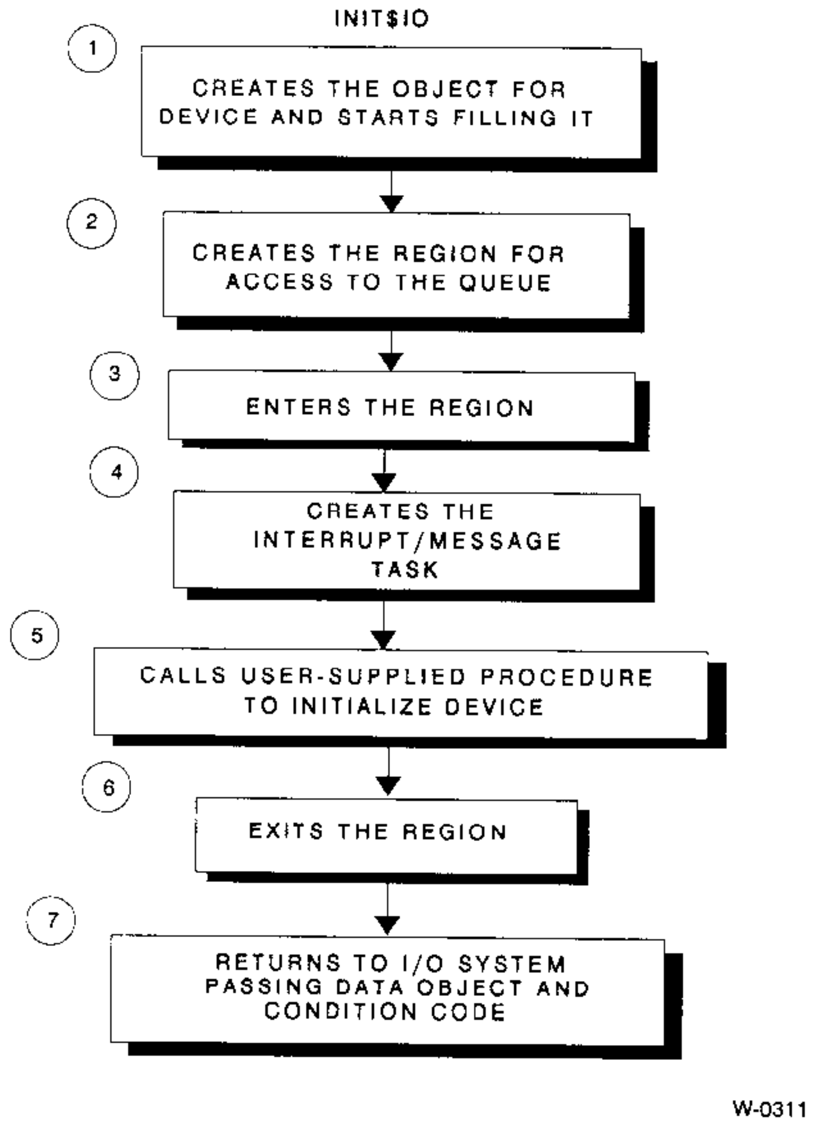

INIT$IO
1
CREATES THE OBJECT FOR DEVICE AND STARTS FILLING IT
2
CREATES THE REGION FOR ACCESS TO THE QUEUE
3
ENTERS THE REGION
4
CREATES THE INTERRUPT/MESSAGE TASK
5
CALLS USER-SUPPLIED PROCEDURE TO INITIALIZE DEVICE
6
EXITS THE REGION
7
RETURNS TO I/O SYSTEM PASSING DATA OBJECT AND CONDITION CODE
W-0311

Figure B-1. Random Access Device Driver Initialize I/O Procedure

1. It creates a data storage area to be used by all the procedures in the device driver. The size of this area depends in part on the number of units in the device and any special space requirements of the device. INIT$IO then begins initializing this area and eventually places the following information there:

   - A token for a region. Step 2 creates this region for mutual exclusion.
   - An array to contain the addresses of the DUIBs for the device-units attached to this device. INIT$IO places the address of the DUIB for the first attaching device unit into this array.
   - A token for the message task.
   - Other values indicating the queue is empty and the driver is not busy.
   - A port object used by the message task to receive messages from the controller. The user-supplied driver uses this object to send messages to the controller.

   It also reserves space in the data storage area for device data.

2. It creates a region. The other random access support routines receive control of this region whenever they place a request on the queue or remove a request from the queue. INIT$IO places the token for this region in the data storage area.
3. It enters the region to prevent the message task from starting before initialization is complete.
4. It calls a device-specific device initialization procedure that initializes the device itself. It gets the address of this procedure by examining the Device Information Table specified in the DUIB. Refer to Chapter 5 for information on how to write this initialization procedure.
5. It creates a message task to handle messages generated by this device. When INIT$IO invokes CREATE$TASK to create the message task, it does not specify the task's data segment. Instead, it uses the data$seg parameter of CREATE$TASK to pass the message task a token for the data storage area. This area is where the message task will get information about the device. INIT$IO places the actual data segment value, as well as a token for the message task, in the data storage area.

6. It exits the region.
7. It returns control to the I/O System, passing a token for the data storage area and a condition code which indicates the success of the initialization operation.

If an error occurs at any point in these steps, the INIT$IO procedure exits the region, deletes all the objects it has created up to that point, and returns an error to the I/O System.

B.2 FINISH$IO PROCEDURE

The I/O System calls FINISH$IO when an application task makes an RQ$A$PHYSICAL$DETACH$DEVICE system call and no other units of the device are currently attached.

FINISH$IO calls a user-supplied procedure to perform final processing on the device, deletes the message task, and deletes the objects used by the other device driver routines.

When the I/O System calls FINISH$IO, it passes the following parameters:

- A pointer to the DUIB of the device-unit just detached
- A TOKEN for the data storage area created by INIT$IO

The following paragraphs show the general steps the FINISH$IO procedure follows to terminate processing for a device. Figure B-2 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It calls a device-specific device finish procedure that performs any necessary final processing on the device itself. FINISH$IO gets the address of this procedure by examining the Device Information Table specified in the DUIB. Refer to Chapter 5 for information about device information tables.
2. It deletes the message task originally created for the device by the INIT$IO procedure.
3. It deletes the region and the data storage area originally created by the INIT$IO procedure, allowing the Operating System to reallocate the memory used by these objects.
4. It returns control to the I/O System.

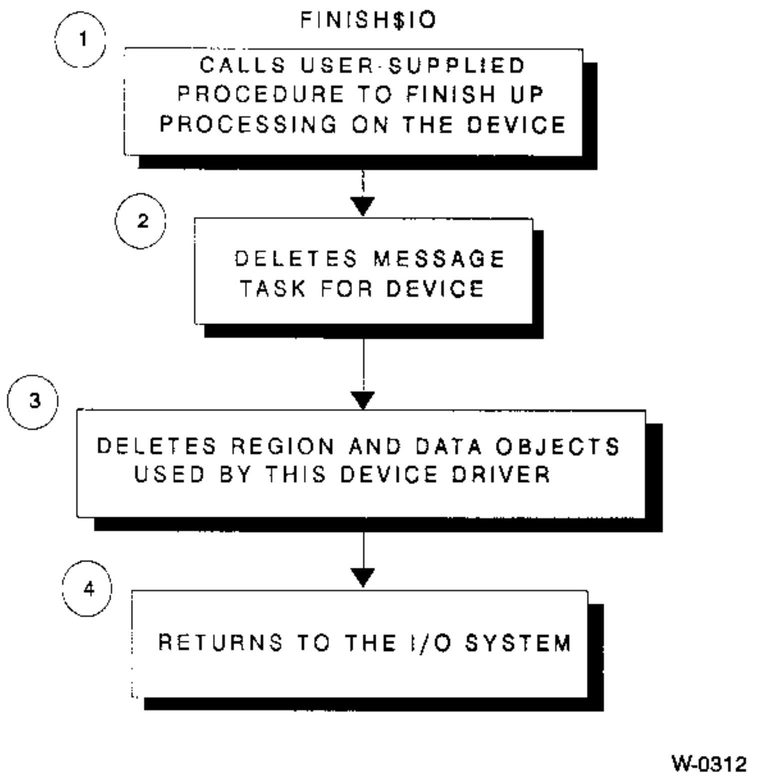

FINISH$IO
1
CALLS USER-SUPPLIED PROCEDURE TO FINISH UP PROCESSING ON THE DEVICE
2
DELETES MESSAGE TASK FOR DEVICE
3
DELETES REGION AND DATA OBJECTS USED BY THIS DEVICE DRIVER
4
RETURNS TO THE I/O SYSTEM
W-0312

Figure B-2. Random Access Device Driver Finish I/O Procedure

B.3 QUEUE$IO PROCEDURE

For message-passing devices, the I/O System calls the QUEUE$IO procedure to place an I/O request on a queue of requests on a first-in-first-out basis. This queue has the structure of the doubly-linked list shown in Figure 7-1. This procedure calls a user-supplied procedure to start processing the I/O requests.

When the I/O System calls QUEUE$IO, it passes the following parameters:

- A token for the IORS
- A pointer to the DUIB
- A token for the data storage area originally c eated by INIT$IO

The following paragraphs show the general steps the QUEUE$IO procedure goes through to place a request on the I/O queue. Figure B-3 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It sets the DONE field in the IORS to 0H, indicating tne request has not yet been completely processed. Other procedures that start the I/O transfers and provide message handling also examine and set this field. It also sets IORS.STATUS to E$OK and IORS.ACTUAL to 0H.
2. It receives control of the region and thus access to the queue. This allows QUEUE$IO to adjust the queue without concern that other tasks might also be doing this at the same time.
3. It verifies that the request is within the range of zero to device size for this device. If the request is outside this range, QUEUE$IO returns E$PARAM. Then it places the IORS on the queue.
4. QUEUE$IO calls the device-specific device start procedure to process the request at the head of the queue. This start procedure is described in Chapter 5.
5. It surrenders control of the region, thus allowing other routines to have access to the queue.
6. It returns control to the I/O System.

NOTE

If the request is complete, QUEUE$IO returns the IORS to the response mailbox; if not, the message task returns it upon completion. The random access support does not return a CLOSE request until all prior requests for the same unit are completed.

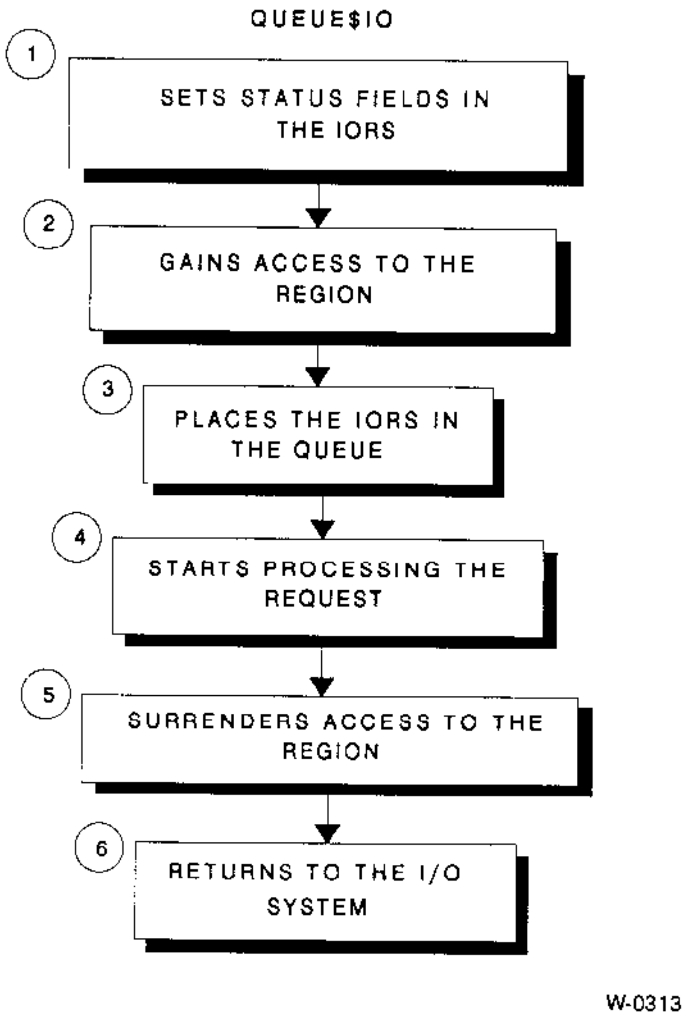

QUEUE$IO
1
SETS STATUS FIELDS IN THE IORS
2
GAINS ACCESS TO THE REGION
3
PLACES THE IORS IN THE QUEUE
4
STARTS PROCESSING THE REQUEST
5
SURRENDERS ACCESS TO THE REGION
6
RETURNS TO THE I/O SYSTEM
W-0313

Figure B-3. Random Access Device Driver Queue I/O Procedure

B.4 CANCEL$IO PROCEDURE

This procedure performs no operations for message-passing devices. The message task sweeps through the request queue and starts all requests. Because of this feature, all I/O requests are guaranteed to finish within a limited time.

NOTE

The CLOSE request supplied by the I/O System is immediately sent to the user-supplied device start procedure. However, the random access support does not return it to the I/O System until all requests in the queue have been completed.

B.5 MESSAGE TASK (MESSAGE$TASK)

The INIT$IO procedure creates a message task for the entire device. This message task responds to all messages generated by the units of the device, processes those messages, and starts the device working on the unstarted I/O requests on the queue.

The following paragraphs show the general steps the message task for the random access device driver follows to process a message from the device. Figure B-4 illustrates these steps. The numbers in Figure B-4 correspond to the step numbers in the text.

1. It uses the contents of the processor's DS register to obtain a token (identifier) for the device data storage area. This is possible for these reasons:

    - When INIT$IO created the message task, instead of specifying the message task's DS register in the data$seg parameter of the CREATE$TASK call, it passed the token of the data storage area in this parameter. Therefore, when the Nucleus created the task, it set the task's DS register to the value of the token.

    - When the INIT$IO procedure initialized the data storage area, it included the value of the message task's DS register there.

    When the message task starts running, it saves the contents of the DS register (to use as the address of the data storage area) and sets the DS register to the value listed in the data storage area. Thus the DS register does point to the task's data segment, and the task also knows the address of the data storage area. This is the mechanism used to pass the address of the device's data storage area from the INIT$IO procedure to the message task.

2. It begins an infinite loop by invoking the RQ$SEND$RSVP call to wait at the port for messages from the device. For more information on how to send and receive messages to/from specific ports, see the description of the Nucleus Communications Service in the *Extended iRMX II Nucleus User's Guide.* For setting the message task priority, see the description of the "Nucleus Communications" screen in the *Extended iRMX II Interactive Configuration Utility Reference* manual.

3. Via a region, it gains access to the request queue. This allows it to examine the first entry in the request queue without concern that other tasks are modifying it at the same time.

4. It calls a user-written device-interrupt procedure to process the received message. This interrupt procedure is described further in Chapter 5.

5. The message task checks the status of the next request in the queue.
6. If the request has been completely processed, (one request can require multiple reads or writes, for example), the message task removes the IORS from the queue and sends it as a message to the response mailbox (exchange) indicated in the IORS. If the request is not completely processed, the message task leaves the IORS in the queue but checks to see if the request has been started.
7. If the request has not been started, the message task calls the user-supplied device start procedure to process the request.
8. In any case, the message task then surrenders access to the queue, allowing other routines to modify the queue, and loops back to wait for another message from the controller.

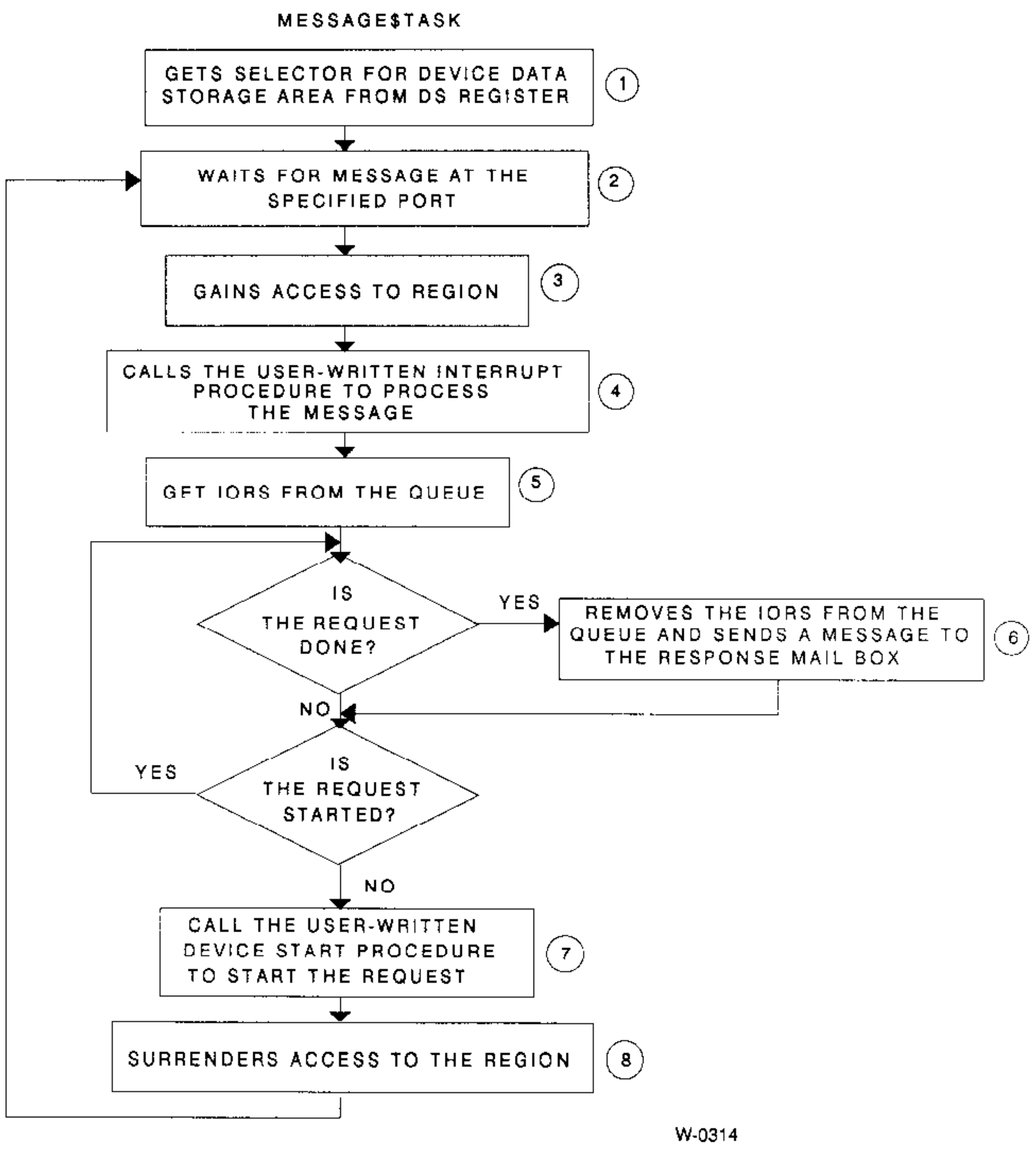

MESSAGE$TASK
GETS SELECTOR FOR DEVICE DATA STORAGE AREA FROM DS REGISTER
1
WAITS FOR MESSAGE AT THE SPECIFIED PORT
2
GAINS ACCESS TO REGION
3
CALLS THE USER-WRITTEN INTERRUPT PROCEDURE TO PROCESS THE MESSAGE
4
GET IORS FROM THE QUEUE
5
IS THE REQUEST DONE?
YES
REMOVES THE IORS FROM THE QUEUE AND SENDS A MESSAGE TO THE RESPONSE MAIL BOX
6
NO
IS THE REQUEST STARTED?
YES
NO
CALL THE USER-WRITTEN DEVICE START PROCEDURE TO START THE REQUEST
7
SURRENDERS ACCESS TO THE REGION
8
W-0314

Figure B-4. Random Access Device Driver Message Task

APPENDIX C SUPPORTING THE STANDARD DISKETTE FORMAT

For compatibility with ECMA (European Computer Manufacturers Association) and ISO (International Organization for Standardization), the Intel-supplied device drivers can format the beginning tracks of all flexible diskettes in the same manner, regardless of the format of the remainder of the diskette. This formatting is referred to as standard formatting. The other option, in which all tracks of a diskette have the same format, is referred to as uniform formatting.

The standard formatting for cylinder 0 on flexible diskettes is as follows:

<u>For 5-1/4" diskettes</u>

- Cylinder 0, side 0 is formatted with 128-byte sectors, single density, 16 sectors per track.
- If the diskette is double-sided, cylinder 0, side 1 is formatted like the rest of the tracks on the diskette.

<u>For 8" diskettes</u>

- Cylinder 0, side 0 is formatted with 128-byte sectors, single density, 26 sectors per track.
- If the diskette is double-sided, cylinder 0, side 1 is formatted with 256-byte sectors, double density, 26 sectors per track.

The FLAGS field in a device's DUIB indicates whether that device expects (reads, writes, and formats) diskettes in standard or uniform format.

To be consistent with the Intel-supplied drivers, and to be able to correctly access standard format diskettes from other systems, random access diskette drivers that you write must be able to read, write, and format diskettes in this standard format.

To access standard-formatted diskettes, a device driver must be able to translate a logical block number (as supplied to it in the DEV$LOC field of the IORS by the I/O System) into a physical address (cylinder, head, and sector). It must take into consideration that track 0 might be formatted differently than the rest of the diskette, and that there might be a different number of logical blocks on track 0.

The following algorithm can be used to calculate the physical address for 5-1/4" flexible diskette requests. It assumes the program has access to the IORS and the DUIB. A similar algorithm can be used for 8" diskettes. (Remember, the algorithm for 8" diskettes must also take into account the special formatting of cylinder 0, side 1 on double-sided diskettes.)

```
/*  Calculate the number of logical blocks on the standard-formatted
 *  track 0 using the standard granularity and standard number of sectors
 *  per track.
 */

track-0-blocks =    (128 bytes/sector x 16 sectors/track)
     (device-granularity in bytes/sector)

/*  Calculate the number of blocks missing from track 0 (those that would
 *  be there if the diskette were uniformly formatted).  The normal track
 *  size equals the number of sectors per track on the rest of the disk
 *  (obtained from the driver-specific unit information table).
 */

track-0-blocks-missing = normal-track-size - track-0-blocks

/*  If the logical block number of this request indicates a track 0
 *  request, calculate the address.
 */

IF block-number < track-0-blocks  THEN
DO

    /*  Set the cylinder and head number to 0 because this is track 0
     */

    cylinder-num = 0
    head-num = 0

    /*  Add 1 to this equation because diskette sectors start at
     *  1, not 0
     */

    sector-num =    (block-number x device-granularity)  + 1
         (128 bytes/sector)

    /*  See if the request goes beyond track 0
     */

    IF  (bytes-requested)  >  (track-0-blocks  -  block-number)  THEN
        (device-granularity)
    DO

        /*  If the request goes beyond track 0, then calculate the number
         *  of bytes to read or write that are past track 0.  Save the
         *  number until track 0 operations are complete.  Then use the
         *  number to complete the read or write operation.
         */

        remainder = bytes-requested - (track-0-blocks - block-number)
            x device-granularity
    END

    /*
     *  Calculation of physical address is complete for requests that
     *  access track 0.
     */
    RETURN
END

ELSE
DO

    /*
     *  If the request is past track 0, adjust the block number for this
     *  request by adding the number of logical blocks missing from track
     *  0 and calculating the cylinder, head, and sector as if this were
     *  a uniformly-formatted flexible disk.
     */

    adjust-block-num = block-number + track-0-blocks-missing

    /*
     *  First calculate the cylinder number of this request
     */

    cylinder-num =          adjust-block-num
        (total-num-of-heads x track-size)

    /*
     *  Next calculate the head number
     */

    IF total-num-of-heads = 1 THEN
    DO
        /*
         *  This is a one-sided flexible diskette
         */

        head-num = 0
    END
```

```
        ELSE
        DO
            /*
             *  This is a double sided flexible diskette
             */

            temp = adjust-block-num MOD (track-size x 2)
            head-num =     temp
                ________________
                track-size
        END
        /*
         *  Finally, calculate sector number for this request, adding 1
         *  because flexible diskette sectors start at 1.
         */

        sector-num = temp MOD track-size + 1
    END
```

Intel®

APPENDIX D INTERPRETING BAD TRACK INFORMATION

Hard disk drives obtained from Intel (either separately or as part of complete systems) have recorded in special locations information about which tracks or which sectors of the disk are unreliable and should not be used. When these hard disks are used with staIntel iSBC 214, iSBC 215G, and iSBC 220 controllers, Intel-supplied device drivers can read this bad track information and map out the unreliable areas when formatting the disk.

With the iSBC 214, iSBC 215G, and iSBC 220 controllers supported by the I/O System, the information on the disk must refer to entire tracks that are unreliable. The iSBC 214, iSBC 215G, and iSBC 220 controllers format disks a track at a time, and therefore are capable of mapping out only entire tracks.

To assist in adding this capability to the drivers you write, this appendix describes the format Intel uses when writing the bad track information. Any hard disk drivers you write should be able to obtain this bad track information and map out the bad tracks whenever they format the disks.

The bad track information is recorded on the highest-numbered cylinder - 1 (the highest-numbered cylinder is reserved for diagnostic tracks).

The last four tracks of that cylinder contain the bad track information. Each track contains the same information but is formatted with a different sector size:

Track	Sector Size
Last cylinder - 1, Last surface	128 bytes/sector
Last cylinder - 1, Last surface - 1	256 bytes/sector
Last cylinder - 1, Last surface - 2	512 bytes/sector
Last cylinder - 1, Last surface - 3	1024 bytes/sector

If a disk has less than four recording surfaces (and therefore less than four tracks per cylinder), the tracks on the next cylinder (last cylinder - 2) are used for the remaining bad track information.

Recording the information in four different sector sizes allows the driver to access the information during format time, regardless of the sector size chosen by the user. For example, if the user decides to format the disk with a volume granularity (sector size) of 512 bytes, the driver sets up the controller for 512-byte sectors and accesses the bad track information from the location (last cylinder - 1, last surface - 2). Likewise, when formatting in 1024-byte sectors, the driver obtains the bad track information from the location (last cylinder - 1, last surface - 3).

On each of those tracks, 1024 bytes of bad track information is recorded four times, starting at sector 0, with a 1024-byte gap between each recording. The multiple occurrences are insurance against bad spots in this area of the disk. If an error occurs when the driver attempts to access the first occurrence of the bad track information, it tries again with the second occurrence, and so forth.

The format of the bad track information is as follows:

WORD	Must contain the value 0ABCDH
WORD	Number of bad tracks in this list (maximum of 255)

Then, for each bad track, the following information appears:

WORD	Cylinder number of bad track
BYTE	Surface number of bad track
BYTE	Set this field to zero

Figure D-1 illustrates the position of this bad track information on the disk.

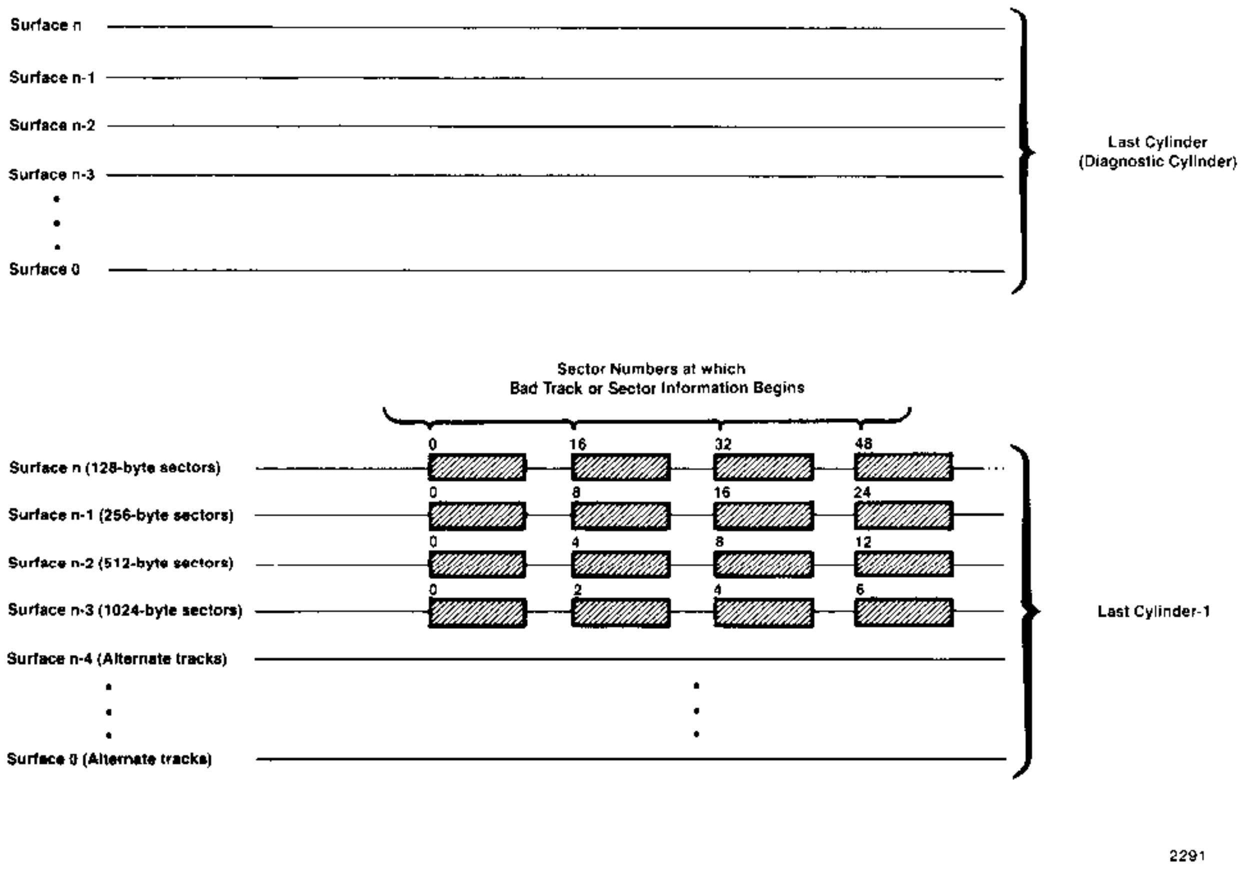

Surface n
Surface n-1
Surface n-2
Surface n-3
Surface 0
Last Cylinder (Diagnostic Cylinder)
Sector Numbers at which Bad Track or Sector Information Begins
Surface n (128-byte sectors)
0 16 32 48
Surface n-1 (256-byte sectors)
0 8 16 24
Surface n-2 (512-byte sectors)
0 4 8 12
Surface n-3 (1024-byte sectors)
0 2 4 6
Surface n-4 (Alternate tracks)
Surface 0 (Alternate tracks)
Last Cylinder-1
2291

Figure D-1. Format of Bad Track Information

82530 terminal driver 3-14
8274 terminal driver 3-13

A

APC sequences 2-13
attach device system calls 8-1
auto-answer modem 2-44
axes sequence and orientation 2-21
axes sequence control 6-13

B

bad track information D-1
bad track/sector information 4-16
bad tracks and sectors 8-11
baud rate
  input 2-21, 6-13
  output 2-21, 6-14
BEGIN$LONG$TERM$OP procedure 5-28
BIOS$GET$ADDRESS procedure 4-21
board ID 5-13
buffer
  line-edit 2-7
  raw input 2-6
  Terminal Support Code 2-7
  type-ahead 2-7
buffered devices 6-20
buffers
  number of 4-6
Byte Bucket driver 3-14

C

cancel I/O procedure 7-1, 4
cancel requests 8-2
CANCEL$IO 5-2
CANCEL$IO procedure 4-5
  interrupt-driven devices A-42
  message-passing devices B-7

character length 6-28
characteristics of diskettes 4-3
close system calls 8-2
closing files 8-5
common drivers 1-4, 2-1, 3-2, 5-1
communication levels 1-1
configuring device drivers 9-1
connection flags 6-9
connection modes 2-16
control character redefinition 2-43
control characters 2-24
  output 2-17
control strings 2-12
CONTROL-O character 2-12
CONTROL-P character 2-10
CONTROL-Q character 2-12
CONTROL-R character 2-10
CONTROL-S character 2-12
CONTROL-T character 2-12
CONTROL-U character 2-10
CONTROL-W character 2-12
CONTROL-X character 2-10
CONTROL-Z character 2-10
cursor addressing offset 2-22
cursor positioning 2-41
custom drivers 1-4, 2-1, 3-14, 7-1
cylinder size 5-15

D

delete character 2-10
deleting lines 2-10
detach device system calls 8-1
device data storage area 5-15
device driver configuration 9-1
device driver interfaces 4-1
device driver types 1-4
device drivers
  Intel-supplied 3-1
device drivers 1-1
device finish procedure 5-11, 1-6, 9
device granularity 4-4
device information screens 9-9
device information table 4-5, 5-8, 6-4, 18
device initialization procedure 5-11, 16, 17

device interfaces 4-21
device interrupt procedure 5-12, 16, 21
device number 1-3
device start procedure 5-12, 16, 19
device stop procedure 5-12, 16, 21
device-unit information block
  creating 4-8
device-unit information block (DUIB) 4-1, 6-3
device-unit information screens 9-16
device-unit number 1-3
discarding mode 2-11
discarding output 2-12
diskette characteristics 4-3
diskette format, standard C-1
drive characteristics 4-14
drive characteristics 8-7
driver interfaces 4-1
drivers
  Intel-supplied 3-1
DUIB
  creating 4-8
DUIB 4-1, 6-3
duplex 2-20, 6-11

E

echo control 6-10
echoing 2-17
emptying type-ahead buffer 2-10
end of file character 2-10
END$LONG$TERM$OP procedure 5-29
escape sequence 2-24, 27, 33
examples
  simulation 2-32
  translation 2-29
explicit seeks 5-15

F

F$ATTACH requests 8-4
F$CLOSE requests 8-5
F$DETACH requests 8-4
F$OPEN requests 8-4
F$READ requests 8-3
F$SEEK requests 8-4

F$SPECIAL requests 8-5
F$WRITE requests 8-3
file drivers 1-1, 4-2
file marks 8-10
finish I/O procedure 4-5, 7-1, 2
FINISH$IO 5-1
FINISH$IO procedure 4-5
  interrupt-driven devices A-39
  message-passing devices B-4
fixed update 4-6
flow control 2-22, 6-26
flush mode 2-8, 17
formatting tracks 4-13, 8-5
FS$FORMAT$TRACK requests 8-5
FS$GET$BAD$INFO requests 8-11
FS$GET$DRIVE$DATA requests 8-7
FS$GET$TERMINAL$ATTRIBUTES requests 8-8
FS$NOTIFY requests 8-6
FS$QUERY requests 8-6
FS$READ$FILE$MARK requests 8-10
FS$RESET requests 8-9
FS$RETENSION$TAPE requests 8-10
FS$SATISFY requests 8-6
FS$SET$BAD$INFO requests 8-11
FS$SET$SIGNAL requests 8-9
FS$SET$TERMINAL$ATTRIBUTES requests 8-8
FS$WRITE$FILE$MARK requests 8-10

G

GET$IORS procedure 5-29
granularity -4

H

high water mark 2-23, 6-27
high water mark, special 2-24

I

I/O request/result segment (IORS) 4-9
I/O requests 1-6, 2-4, 8-1
ICU Merge (ICUMRG) utility 9-2, 9-4, 22
ICUMRG utility 9-2, 4, 22
implied seeks 5-15

INIT$IO 4-4, 5-1
INIT$IO procedure
  interrupt-driven devices A-35
  message-passing devices B-1
initialize I/O procedure 4-4, 7-1
input baud rate 2-21, 6-13
input parity 2-17, 20, 6-10, 11, 28
inserting data into the input stream 2-50
instance 5-13
Intel-supplied device drivers 3-1
interfaces
  device 4-21
interfaces, driver 4-1
interrupt 5-1
interrupt handler 2-3
interrupt level 5-10, 6-7
interrupt lines 6-7
interrupt task 2-3, 5-2, A-45
interrupt type 6-17
INTERRUPT$TASK 5-2, A-45
interrupt-driven devices 5-1, 6-1
introduction 1-1
invoking the UDS utility 9-18
IORS 4-9
iSBC 186/224A multi-peripheral driver 3-6
iSBC 186/410 terminal driver 3-9
iSBC 208 disk driver 3-2
iSBC 214 disk and tape controller 3-5
iSBC 215G disk controller 3-3
iSBC 217C tape controller 3-3
iSBC 220 SMD disk driver 3-6
iSBC 264 bubble memory driver 3-8
iSBC 286/10(A) line printer driver 3-8
iSBC 534 terminal driver 3-11
iSBC 544A terminal driver 3-12
iSBX 218A diskette controller 3-4
iSBX 218A diskette driver 3-5
iSBX 251 bubble memory driver 3-7
iSBX 350 line printer driver 3-8
iSBX 351 terminal driver 3-12

L

levels of communication 1-1
line editing control 6-10

line protocol 2-20
line protocol indicator 6-11
line terminator 2-10
line terminator, special 2-10
line-edit buffer 2-7
line-editing functions 2-9
line-editing mode 2-17
link parameters 6-29
locking the terminal 2-49
logical addresses 4-21
long-term operations 2-4, 5-28
low water mark 2-23, 6-27

M

Mass Storage Controller (MSC) driver 3-2
message 5-1, 5-23, 6-42
message task 5-3, B-8
MESSAGE$TASK B-8
message-passing devices 5-1, 6-1
mode
  flush 217
  line-editing 2-17
  special character 6-25, 30
  terminal 4-15
  transparent 2-8, 17
modem 2-44, 22
modem indicator 2-20
modes
  connection 2-16
  terminal 2-18
MSC driver 3-2
MULTIBUS I interruptless drivers 6-1
MULTIBUS II drivers 6-1

N

normal mode 2-11
notification of drive door open 4-14
NOTIFY procedure 5-26
notify requests 8-6
NUM$BUFFERS 2-2, 4-6

O

open system calls 8-2
opening files 8-4
optimizing seeks 2-5
OSC control 2-17
OSC sequences 2-12, 26, 27, 6-10
output baud rate 2-21, 6-14
output control characters 2-17, 6-10, 23
output medium 2-20, 6-11
output mode 2-11
output parity 2-17, 20, 6-10, 12, 28
overflow offset 2-22
overlapping seeks 2-5

P

parity
  input 2-17, 20, 6-10, 28
  output 2-17, 20, 6-10, 28
physical addresses 4-21
physical link 2-23
physical link parameters 6-27
port 6-1
positioning the cursor 2-41
priority 5-11

Q

query requests 8-6
queue I/O procedure 4-5, 7-1, 3
queue size 5-13
QUEUE$IO 5-2
QUEUE$IO procedure
  interrupt-driven devices A-40
  message-passing devices B-5
QUEUE$IO procedure 4-5
quoting character 2-10

R

RAM driver 3-15
random access drivers 1-4, 2-1, 3-2, 5-1

random access support routines
  interrupt-driven devices A-34
  message-passing devices B-1
raw input buffer 2-6, 6-19
read requests 8-3
read system calls 8-2
redefining control characters 2-43
redisplaying lines 2-10
request queue 2-3, 7-5
retries 2-5
retrying I/O requests 5-14

S

satisfy requests 8-6
screen height 2-21
screen width 2-21
scrolling count 6-20
scrolling mode 2-11, 12
scrolling number 2-21, 6-14
SCSI driver 3-9
seek optimization 2-5
seek overlap 2-5
seek system calls 8-2
SEEK$COMPLETE procedure 5-15, 27
seeking 8-4
seeks
  explicit 5-15
  implied 5-15
signal characters 4-15, 8-9
simulation 2-25, 31
software control strings 2-12
special array 2-25
special character mode 6-25, 26, 30
special characters 2-22
special high water mark 2-24, 6-29
special line terminator 2-10
special system calls 8-2
stack size 5-11, 6-6
standard diskette format C-1
starting output 2-12
stop bits 2-24, 6-28
stopping output 2-12
stuffing data into the input stream 2-50

T

tape drives 4-14
tape file marks 4-15
tape requests 8-10
task
  interrupt 5-2
  message 5-3
terminal answer procedure 6-7, 31, 36
terminal attributes8
terminal character sequences 2-26, 40
terminal check procedure 6-8, 32, 37
Terminal Communications Controller driver 3-10
terminal drivers 1-4, 2-1, 3-9
terminal drivers 6-1
terminal finish procedure 6-6, 31, 34
terminal flags 6-11
terminal hangup procedure 6-7, 32, 37
terminal I/O 2-6
terminal initialization procedure 6-6, 31, 32
terminal mode information 4-15
terminal modes 2-18
terminal mutual exclusion procedure 6-30
terminal output 2-11
terminal output procedure 6-7, 32, 44
terminal set output buffer size procedure 6-30, 31
terminal setup procedure 6-6, 31, 34
Terminal Support Code 6-1, 6-14
Terminal Support Code input buffer 2-7
terminal utility procedure 6-7, 32, 45
track formatting 4-13, 8-5
track size 5-14
translation 2-21, 25, 29, 6-13
transparent mode 2-8, 17
type of terminal interrupt 6-17
type-ahead buffer 2-7
types of device drivers 1-4

U

UDS error messages 9-19
UDS utility 9-2, 4
unit information screens 9-14
unit information table 4-5, 5-13, 6-9, 18
unit number 1-3
unlocking the terminal 2-49
update timeout 4-5
User Device Support (UDS) utility 9-2
User Device Support Utility (UDS) 9-4

V

volume change notification 2-3

W

write requests 8-3
write system calls 8-2

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
Flower-Hill Shin-machi
1-23-9, Shinmachi
Setagaya-ku, Tokyo 15

NETHERLANDS
Intel Semiconductor (Netherland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor G.N.B.H.
Seidlestrasse 27
D-8000 Munchen

intel®

iRMX® II.3
Volume 2
User's Guides
461845-001

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative.

1. Please describe any errors you found in this publication (include page number).

____

____

____

____

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

____

____

____

____

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

____

____

____

____

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

____

____

____

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ____________

NAME ______________________ DATE ____________
TITLE ______________________
COMPANY NAME/DEPARTMENT ______________________
ADDRESS ______________________ PHONE (   ) ____________
CITY ____________ STATE ____________ ZIP CODE ____________
(COUNTRY)

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.

If you are in the United States, use the preprinted address provided on this form to return your comments. No postage is required. If you are not in the United States, return your comments to the Intel sales office in your country. For your convenience, a list of international sales offices is provided directly before this mailer.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
OMO SW Technical Publications, MS: HF3-60
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-9987