intel®

iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART II

For Release 6

iRMX™ 86 OPERATING SYSTEM

Copyright © 1984, Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 146196-001

ASSEMBLY INSTRUCTIONS

Volume: iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART II
Order No: 146196-001

INTRODUCTION

This sheet describes how to assemble this iRMX 86 literature packet. The assembly is simple and takes less than 5 minutes.

This literature packet contains:

- The literature in the volume, including this instruction sheet and these manuals:
  - iRMX 86 APPLICATION LOADER REFERENCE MANUAL
  - iRMX 86 HUMAN INTERFACE REFERENCE MANUAL
  - iRMX 86 UNIVERSAL DEVELOPMENT INTERFACE REFERENCE MANUAL
  - GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O SYSTEMS
  - iRMX 86 PROGRAMMING TECHNIQUES
  - iRMX 86 TERMINAL HANDLER REFERENCE MANUAL
  - iRMX 86 DEBUGGER REFERENCE MANUAL
  - iRMX 86 SYSTEM DEBUGGER REFERENCE MANUAL
  - iRMX 86 CRASH ANALYZER REFERENCE MANUAL
  - iRMX 86 BOOTSTRAP LOADER REFERENCE MANUAL
- The first of two cardboard separators.
- Ten divider tabs, one for each manual.
- The bottom cardboard separator.

If your binder package is missing one or more of these items, contact Intel immediately.

ASSEMBLY

Assembling the volume involves inserting the literature packet into its three-ring binder and placing an appropriately labeled divider tab at the front of each manual in the volume.

At this point you have torn open the shrink wrapping, removed the entire literature packet, and extracted this sheet from the packet. Set this sheet aside. You will be referring to it as you go.

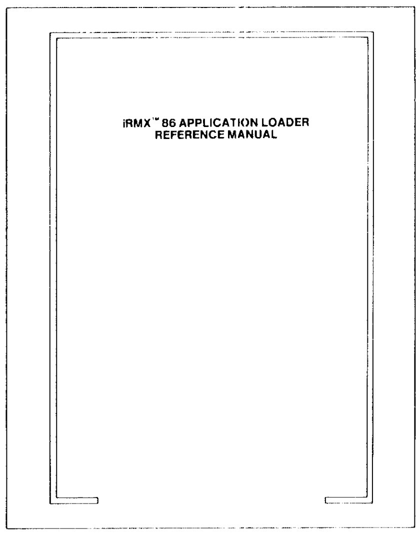
To put the volume together, follow these steps:

(over)

1. Separate the divider tabs from the rest of the literature packet. Discard the cardboard. The divider tabs have these labels and match these manuals:

Label	Manual
Application Loader	iRMX 86 APPLICATION LOADER REFERENCE MANUAL
Human Interface	iRMX 86 HUMAN INTERFACE REFERENCE MANUAL
UDI	iRMX 86 UNIVERSAL DEVELOPMENT INTERFACE REFERENCE MANUAL
Device Drivers	GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O SYSTEMS
Programming Techniques	iRMX 86 PROGRAMMING TECHNIQUES
Terminal Handler	iRMX 86 TERMINAL HANDLER REFERENCE MANUAL
Debugger	iRMX 86 DEBUGGER REFERENCE MANUAL
Crash Analyzer	iRMX 86 CRASH ANALYZER REFERENCE MANUAL
System Debugger	iRMX 86 SYSTEM DEBUGGER REFERENCE MANUAL
Bootstrap Loader	iRMX 86 BOOTSTRAP LOADER REFERENCE MANUAL

2. Find Page xiv, which is at the end of the Volume Contents. Open the binder rings and insert the Front Cover up to and including Page xiv into the left side of the open rings. The top page of the literature packet is now the "Application Loader" title page, which looks like this:



3. Insert the divider tab labeled "Application Loader" into the left side of the open rings.

4. Insert the text of the Application Loader manual into the left side of the binder rings. The last page of the Application Loader manual is "Application Loader Index-2." The top page of the literature packet should now be the title page of the Human Interface manual.

5. Repeat the process for the remaining manuals, matching divider tabs with manuals.

6. Close the binder rings. Discard the shrink wrapping and this instruction sheet.

***

iRMX™ 86
PROGRAMMER'S REFERENCE MANUAL, PART II
For Release 6

Order Number: 146196-001

Copyright © 1984, Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

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

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

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

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

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

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

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

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

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

BITBUS	iLBX	iPDS	Plug-A-Bubble
COMMputer	im	iRMX	PROMPT
CREDIT	iMMX	iSBC	Promware
Data Pipeline	Insite	iSBX	QUEX
GENIUS	Int$_{e}$l	iSDM	QUEST
$\overset{\triangle}{\mathrm{i}}$	int$_{e}$l	iSXM	Ripplemode
i	int$_{e}$lBOS	Library Manager	RMX/80
I^{2}ICE	Intelevision	MCS	RUPI
ICE	int$_{e}$ligent Identifier	Megachassis	Seamless
iCS	int$_{e}$ligent Programming	MICROMAINFRAME	SOLO
iDBP	Intellec	MULTIBUS	SYSTEM 2000
iDIS	Intellink	MULTICHANNEL	UPI
	iOSP	MULTIMODULE	

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

*MULTIBUS is a patented Intel bus.

Copyright © 1983, Intel Corporation

REV.	REVISION HISTORY	DATE
-001	*Original Issue. Supplies and updates information formerly contained in the iRMX 86 Loader Reference Manual,* the *iRMX 86 Human Interface Reference Manual,* the *Guide to Writing Device Drivers for the iRMX 86 and iRMX 88 I/O Systems,* the *iRMX 86 Programming Techniques,* the *iRMX 86 Terminal Handler Reference Manual,* the *iRMX 86 Debugger Reference Manual,* the *iRMX 86 System Debug Monitor Reference Manual,* and the *iRMX 86 Crash Analyzer Reference Manual.*	3/84

VOLUME PREFACE

This volume, the iRMX 86 PROGRAMMER'S REFERENCE MANUAL, PART II, contains detailed information about iRMX 86 Operating System programming utilities and advanced programming techniques for writing application and system programs.

MANUALS IN THIS VOLUME

This section briefly describes each iRMX 86 manual in the order they appear in this volume.

iRMX™ 86 APPLICATION LOADER REFERENCE MANUAL

Tab Label: Application Loader

This manual describes the iRMX 86 Application Loader, which loads code from secondary storage into RAM for execution. The manual contains detailed descriptions of the system calls available with the Application Loader.

iRMX™ 86 HUMAN INTERFACE REFERENCE MANUAL

Tab Label: Human Interface

This manual documents system calls used to retrieve and interpret the constituent parts of commands entered at a keyboard terminal. This manual describes the system calls used for parsing and processing commands, and for high-level I/O operations to a terminal.

iRMX™ 86 UNIVERSAL DEVELOPMENT INTERFACE REFERENCE MANUAL

Tab Label: UDI

This manual outlines general programming considerations for using the Universal Development Interface (UDI) and describes in detail the UDI system calls in the iRMX 86 Operating System.

GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX™ 86 AND iRMX™ 86 I/O SYSTEMS

Tab Label: Device Drivers

This manual shows how to write device drivers that can be incorporated into the iRMX 86 Operating System. This applies to devices for which the iRMX 86 Operating System does not already supply device drivers.

iRMX™ 86 PROGRAMMING TECHNIQUES

Tab Label: Programming Techniques

This manual provides a number of programming techniques that can reduce the time spent designing and implementing an iRMX 86-based application system. It includes discussions on PL/M-86 size controls, interface procedures, assembly language programming, inter-job communication, and stack sizes.

iRMX™ 86 TERMINAL HANDLER REFERENCE MANUAL

Tab Label: Terminal Handler

This document describes the iRMX 86 Terminal Handler, which provides basic character echoing and line editing functions. The Terminal Handler is typically used with iRMX 86 Operating Systems that do not include the Basic I/O System.

iRMX™ 86 DEBUGGER REFERENCE MANUAL

Tab Label: Debugger

This manual describes the Dynamic Debugger, an interactive debugging tool used with the iRMX 86 Operating System. The Debugger is especially useful because it is "sensitive" to iRMX 86 objects, and it lets you debug one or more tasks while the rest of the system continues to run. The manual includes descriptions of iRMX 86 Debugger commands.

iRMX™ 86 SYSTEM DEBUGGER REFERENCE MANUAL

Tab Label: System Debugger

This manual describes the System Debugger (SDB), a static debugging tool that is useful in diagnosing system crashes and other "freeze" situations. The System Debugger, like the Dynamic Debugger, is attuned to iRMX 86 objects. The SDB is an extension of the iSDM 86 and 286 System Debug Monitors. The manual includes descriptions of System Debugger commands.

iRMX™ 86 CRASH ANALYZER REFERENCE MANUAL

Tab Label: Crash Analyzer

This manual describes the iRMX 86 Crash Analyzer, a utility used to produce post-mortem memory dumps and to print a formatted display. The display utility shows iRMX 86 objects (memory segments, tasks, jobs, etc.) along with the state of each object at the time the system failed.

iRMX™ 86 BOOTSTRAP LOADER REFERENCE MANUAL

Tab Label: Bootstrap Loader

This manual describes the iRMX 86 Bootstrap Loader, a start-up utility that loads user-selected code into memory for execution after system reset.

iRMX™ 86 PUBLICATIONS

Because the iRMX 86 documentation set is packaged in bound volumes, you can no longer order manuals individually. Instead, you must order a complete volume to get a manual contained in that volume. (Individual manuals no longer have order numbers.)

When ordering a volume, use the order number that appears on the spine of the binder. This number is also provided in the following list. A second number appears on the inside front cover of each volume, but it can be ignored because it is a manufacturing part number used internally at Intel.

The following list shows volume titles, order numbers, and individual manuals in each of the volumes. Manuals are listed in the order they appear in the volumes. This volume is indicated by boldface type.

1. iRMX™ 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL
   Order Number: 146545-001

   - Introduction to the iRMX™ 86 Operating System
   - iRMX™ 86 Operator's Manual
   - iRMX™ 86 Disk Verification Utility Reference Manual

2. iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART I
   Order Number: 146546-001

   - iRMX™ 86 Nucleus Reference Manual
   - iRMX™ 86 Basic I/O System Reference Manual
   - iRMX™ 86 Extended I/O System Reference Manual

3. **iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART II**
   Order Number: 146547-001

   - **iRMX™ 86 Application Loader Reference Manual**
   - **iRMX™ 86 Human Interface Reference Manual**
   - **iRMX™ 86 Universal Development Interface Reference Manual**
   - **Guide to Writing Device Drivers for the iRMX™ 86 and iRMX™ 88 I/O Systems**
   - **iRMX™ 86 Programming Techniques**
   - **iRMX™ 86 Terminal Handler Reference Manual**
   - **iRMX™ 86 Debugger Reference Manual**
   - **iRMX™ 86 Crash Analyzer Reference Manual**
   - **iRMX™ 86 System Debugger Reference Manual**
   - **iRMX™ 86 Bootstrap Loader Reference Manual**

4. iRMX™ 86 INSTALLATION AND CONFIGURATION GUIDE
   Order Number: 146548-001

   - iRMX™ 86 Installation Guide
   - iRMX™ 86 Configuration Guide
   - Master Index for Release 6 of the iRMX™ 86 Operating System

RELATED PUBLICATIONS

- iAPX 86,88 Family Utilities User's Guide, Order Number: 121616
- iAPX 86,88 User's Manual, Order Number: 210201
- PL/M-86 User's Guide, Order Number: 121636
- 8086 Relocatable Object Module Formats, Order Number: 121748
- iSDM™ 86 System Debug Monitor Reference Manual, Order Number: 146165
- iSDM™ 286 System Debug Monitor Reference Manual, Order Number: 145804

- ICE™-86/ICE™-88 Microsystems In-Circuit Emulator Operation Instructions for ISIS-II Users Manual, Order Number: 162554
- iMMX™ 800 MULTIBUS Message Exchange Reference Manual, Order Number: 144912
- iRMX™ 80/88 Interactive Configuration Utility User's Guide, Order Number: 142603
- ASM86 Language Reference Manual for 8080/8085-Based Development Systems, Order Number: 121703
- ASM86 Macro Assembler Operating Instructions for 8086-Based Development Systems, Order Number: 121628

VOLUME CONTENTS

VOLUME CONTENTS (continued)

VOLUME CONTENTS (continued)

VOLUME CONTENTS (continued)

***

iRMX™ 86 APPLICATION LOADER REFERENCE MANUAL

CONTENTS

***

CHAPTER 1 INTRODUCTION TO THE APPLICATION LOADER

The Application Loader is a part of the Operating System, and is used to load programs under the control of iRMX 86 tasks -- tasks that are part of the Operating System, and tasks that are part of applications programs you write.

The Loader provides system calls that load programs from secondary storage into memory. The Loader system calls give you several advantages. They allow programs to run in systems that haven't enough memory to accommodate all of their programs at one time. They allow programs that are seldom used to reside on secondary storage rather than in primary memory. Finally, they make it easier for you to add new programs to the system.

Also, the Loader allows you to implement large programs by using overlays. For example, suppose that your application system includes a large compiler. By dividing the compiler into several parts, you can avoid keeping the entire compiler in RAM. One of the parts, called the root, remains in RAM as long as the compiler is running. The root uses the Loader to load the other parts, called overlays.

This chapter is designed to help you understand the capabilities of the Loader by providing you with background information. The chapter consists of five main parts:

- Loader terminology
- Loader features
- Configuration options
- Preparing code for loading
- How the Loader works

After reading this chapter, you should be able to understand the system call descriptions in Chapter 2.

LOADER TERMINOLOGY

Before attempting to read about the system calls of the Loader, you must become familiar with the terminology used to describe them. The following terms are used fairly frequently in describing system calls:

- object code, object module, and object file
- absolute code, position-independent code (PIC), and load-time locatable code (LTL)

- fixup
- synchronous system calls, and asynchronous system calls
- I/O job
- overlay, root module, and overlay module

The following sections define these terms or refer you to documents in which you can find definitions.

OBJECT CODE

The term object code is used to distinguish between the program that goes into a translator (compiler or an assembler) and the program that comes out of a translator. However, in this manual, object code refers to the following three categories of code:

- output of a translator
- output of the LINK86 command
- output of the LOC86 command

An object module is the output of a single compilation, a single assembly, or a single invocation of the LINK86 or LOC86 commands, and an object file is a named file in secondary storage that contains object code in one or more modules.

TYPES OF OBJECT CODE

The Loader can load absolute code, position-independent code, and load-time-locatable code. These are defined here.

Absolute Code

Absolute code, and an absolute object module, is code that has been processed by LOC86 to run only at a specific location in memory. The Loader loads an absolute object module only into the specific location the module must occupy.

Position-Independent Code (PIC)

Position-independent code (commonly referred to as PIC) differs from absolute code in that PIC can be loaded into any memory location. The advantage of PIC over absolute code is that PIC does not require you to reserve a specific block of memory. When the Loader loads PIC, it obtains iRMX 86 memory segments from the pool of the calling task's job and loads the PIC into the segments.

A restriction concerning PIC is that, as in the PL/M-86 COMPACT model of segmentation (described later in this chapter), it can have only one code segment and one data segment, rather than letting the base addresses of these segments, and therefore the segments themselves, vary dynamically. This means that PIC programs are necessarily less than 64K bytes in length.

PIC code can be produced by means of the BIND control of LINK86.

Load-Time-Locatable (LTL) Code

Load-time locatable code (commonly referred to as LTL code) is the third form of object code. LTL code is similar to PIC in that LTL code can be loaded anywhere in memory. However, when loading LTL code, the Loader changes the base portion of pointers so that the pointers are independent of the initial contents of the registers in the microprocessor. Because of this <u>fixup</u> (adjustment of base addresses), LTL code can be used by tasks having more than one code segment or more than one data segment. This means that LTL programs may be more than 64K bytes in length. FORTRAN 86 and Pascal 86 automatically produce LTL code, even for short programs.

LTL code can be produced by means of the BIND control of LINK86.

SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

A <u>synchronous system call</u> is one in which the calling task cannot continue running while the invoked system call is running. For example, if a task invokes a synchronous Loader system call, the calling task will resume running only after the loading operation has either failed or succeeded.

An <u>asynchronous system call</u> is one in which the calling task can run concurrently with the invoked system call. For a detailed explanation of the behavior of asynchronous system calls, refer to Appendix C.

I/O JOB

An I/O job is a special type of job for tasks that perform I/O using the Extended I/O System. In fact, if a task is not in an I/O job, it cannot successfully use all of the system calls in the Extended I/O System.

The notion of an I/O job relates to the Loader because some of the system calls provided by the Loader use the Extended I/O System. Specifically, the A$LOAD$IO$JOB and the S$LOAD$IO$JOB system calls can be invoked only by tasks running in an I/O job.

If you are unfamiliar with I/O jobs, refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a definition.

OVERLAY

The term "overlay," when used as a verb, refers to the process of loading object code that generally resides in RAM only for short periods of time. For example, suppose that you are building a compiler that is very large. You can design the compiler in either of the following ways:

- The compiler can be structured as a monolithic program that resides on secondary storage until it is needed. Once needed, the entire collection of object code must be loaded into RAM.

- If the compiler is an overlaid program, pieces (overlays) of the compiler reside on secondary storage; individual overlays are loaded as they are needed. In this way, the compiler can run in a much smaller area of memory. Note that the compiler might be slower if it uses overlays, depending upon how it uses the time when the overlays are being loaded.

In order to implement an overlaid program using the Loader, you divide the program into two kinds of modules -- a root module, and one or more overlay modules.

A root module is an object module that controls the loading of overlays. Let's again use an overlaid compiler as an example. Suppose that you are developing an application system incorporating the compiler. When the compiler is invoked, your application system can load the root module of the compiler using A$LOAD$IO$JOB or S$LOAD$IO$JOB. (These system calls are described in the next chapter.) The root module can then use the S$OVERLAY system call to load overlay modules as they are needed.

For more information regarding the notion of overlays, root module, and overlay module, refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE.

LOADER FEATURES

The iRMX 86 Loader provides several features that make it valuable in any application system that loads programs from secondary storage into RAM. Some of these features are:

- Device Independence
- Synchronous and Asynchronous System Calls
- Support for Overlaid Programs
- Configurability

The following sections briefly discuss each of these features.

DEVICE INDEPENDENCE

The Loader can load object code from any device if the device supports iRMX 86 named files and an iRMX 86-compatible device driver is available for it. See the iRMX 86 CONFIGURATION GUIDE for a complete list of devices for which Intel supplies device drivers. If you wish to load from a device for which Intel does not yet supply a device driver, you can write your own device driver. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O SYSTEMS for directions.

SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

The Loader provides you with both synchronous system calls 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. On the other hand, if you prefer ease of use to explicit control, you can use synchronous system calls.

SUPPORT FOR OVERLAID PROGRAMS

The Loader contains a system call that is explicitly designed to simplify the process of loading overlay modules. By using the S$OVERLAY system call, your root module can easily load overlay modules contained in the same object file as the root module.

CONFIGURABILITY

The Loader is configurable. You can select the features of the Loader that your application system needs. If you don't need all of the capabilities of the Loader, you can leave out some options and use a smaller, faster version of it. Configurable features are summarized in Chapter 3 and are discussed in detail in the iRMX 86 CONFIGURATION GUIDE.

PREPARING CODE FOR LOADING

Two factors govern the methods you must use to prepare code for loading. They are:

- The PL/M-86 model of segmentation to which you are adhering.
- Whether you want the loaded calls to be able to invoke iRMX 86 system calls.

In addition to these factors, you must ensure that the object code specifies an entry point and deals with stack size. The following sections address these issues.

PL/M-86 MODELS OF SEGMENTATION AND TYPES OF OBJECT CODE

When you compile your source code, you must (explicitly or implicitly) specify a PL/M-86 model of segmentation (specified at compile time by the SIZE control). The model you specify affects the kind of object code generated. The purpose of this section is to correlate the model of segmentation with the kind of code generated.

The PL/M-86 programming language offers four models of segmentation: SMALL, MEDIUM, LARGE, and COMPACT. The iRMX 86 Operating System does not support the SMALL model. Do not use it to generate any code that you plan to load with the Loader. Table 1-1 explains what you must (or must not) do, in addition to selecting a model of segmentation, in order to produce object code of a particular type.

For more information regarding models of segmentation and their effect on the iRMX 86 Operating System, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.

INVOKING iRMX™ 86 SYSTEM CALLS

If you want your loadable code to invoke iRMX 86 system calls, you must use LINK86 to link the loadable object modules to the iRMX 86 interface procedures. Refer to the iRMX 86 PROGRAMMING TECHNIQUES manual for details.

ENTRY POINTS

Generally, when your tasks invoke the Loader, the Loader must be able to determine the entry point for the loaded object code. (The entry point, also known as the start address, is the location at which execution is to begin.) The Loader uses this information when creating a job in which the loaded code is to run as a task.

Table 1-1. User Actions Required To Match PL/M-86 Model Of Segmentation With Object Code Type

	Model of Segmentation	
Code Type	Medium or Large	Compact
Absolute Code	Use LINK86 without the BIND control to link code together. Use LOC86 to locate the code absolutely.	Use LINK86 without the BIND control to link code together. Use LOC86 to locate the code absolutely.
Position-Independent Code	Not applicable. That is, you cannot produce PIC using the MEDIUM or LARGE model.	Use LINK86 with the BIND control to link code together. Do not use the INITIAL or DATA statement to initialize a pointer. Do not exceed 64K bytes.
Load-Time Locatable Code	Use LINK86 with the BIND control to link code together. Do not locate with LOC86.	Use LINK86 with the BIND control to link code together. Either use the INITIAL or DATA statement to initialize a pointer or exceed 64K bytes.

Using A Main Module

The easiest way to ensure that your object file contains an entry point is to write your source code as a main module; a main module always contains an entry point. Further, if your code is either PIC or LTL code, it must be a main module.

Writing A Procedure To Be Loaded By The Loader

In certain unusual circumstances there are advantages to writing your source code as a procedure rather than as a main module. Such code will have to be loaded using the A$LOAD system call. The mechanics of this loading method are outlined in the description of A$LOAD in the next chapter.

STACK SIZES

When linking (using the LINK86 command) or locating (using the LOC86 command) your code, you must use the SEGSIZE(STACK(...)) control to assign an appropriate stack size. When linking, you must also use the MEMPOOL control if your program issues any Nucleus system calls that create iRMX 86 objects dynamically. The SEGSIZE control is described in the iAPX 86,88 FAMILY USER'S GUIDE.

HOW THE LOADER WORKS

If the Loader is configured into your system, the root job will create the Loader job during initialization of the system. Once created, the Loader job initializes the Loader code and then deletes itself. The Loader code then remains in memory, where it executes as a task whenever a Loader system call is invoked.

***

CHAPTER 2
APPLICATION LOADER SYSTEM CALLS

This chapter describes the PL/M-86 calling sequences for the system calls of the Application Loader. The calls are listed alphabetically. For example, A$LOAD precedes A$LOAD$IO$JOB. This shorthand notation is language-independent and should not be confused with the actual form of the PL/M-86 call. The precise format of each call is defined as part of the detailed description.

These iRMX 86 system calls are declared external procedures in the PL/M-86 language. When you write a program in PL/M-86, you use these procedures to invoke the system calls of the Loader.

Although the system calls are described as PL/M-86 procedures, your tasks can invoke these system calls from assembly language. Refer to the iRMX 86 PROGRAMMING TECHNIQUES manual for information about making system calls in assembly language.

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

RESPONSE MAILBOX PARAMETER

Two system calls described in this chapter are asynchronous. These are the A$LOAD and the A$LOAD$IO$JOB system calls. Your task must specify a mailbox whenever it invokes an asynchronous system call. The purpose of this mailbox is to receive a Loader Result Segment.

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, so details about Loader Result Segments are included in descriptions of the A$LOAD and A$LOAD$IO$JOB system calls.

Avoid using the same response mailbox for more than one concurrent invocation of asynchronous system calls. This is necessary because it is possible for the Loader to 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 only one task invokes the calls and the task always obtains the result of one call via RQ$RECEIVE$MESSAGE before making the next call.

CONDITION CODES

The Loader returns a condition code whenever a system call is invoked. If the call executes without error, the Loader returns the code E$OK. If an error occurs, the Loader returns an exception code.

This chapter includes, for each of the Application Loader's system calls, descriptions of the condition codes that the system call can return. The system call chapters in manuals for the other layers of the iRMX 86 Operating System do the same thing for those layers. You can use the condition code information to write code to handle exceptional conditions that arise when system calls fail to perform as expected. See the iRMX 86 NUCLEUS REFERENCE MANUAL for a discussion of condition codes and how to write code to handle them.

CONDITION CODES FOR SYNCHRONOUS SYSTEM CALLS

For system calls that are synchronous (S$LOAD$IO$JOB and S$OVERLAY), the Loader returns a single condition code each time the call is invoked. If your system has an exception handler, it will receive these codes when exceptional conditions occur, depending upon how the exception mode is set.

CONDITION CODES FOR ASYNCHRONOUS SYSTEM CALLS

For system calls that are asynchronous (A$LOAD and A$LOAD$IO$JOB), the Loader returns two condition codes each time the call is invoked. One code is returned after the sequential part of the system call is executed, and the other is returned after the concurrent part of the call is executed. Your task must process these two condition codes separately.

Appendix C describes the sequential and concurrent portions of asynchronous system calls.

Sequential Condition Codes

The Application Loader returns the sequential condition code in the word pointed to by the except$ptr parameter. If your system has an exception handler, it will receive these codes when exceptional conditions occur, depending upon how the exception mode is set.

Concurrent Condition Codes

The Loader returns the concurrent condition code in the Loader Result Segment it sends to the response mailbox. If the code is E$OK, the asynchronous loading operation ran successfully. If the code is other than E$OK, a problem occurred during the asynchronous loading operation, and your task must decide what to do about the problem. Regardless of the exception mode setting for the application, the exception handler is not invoked by concurrent condition codes, so your program must handle it.

SYSTEM CALL DICTIONARY

The following list is a summary of the iRMX 86 Loader system calls, together with a brief description of each call and the page where the description of the call begins.

Name	Description	Type	Page
A$LOAD	Loads object code or data into memory.	Asynchronous	2-4
A$LOAD$IO$JOB	Creates an I/O job, loads the job's code, and causes the job's task to run.	Asynchronous	2-15
S$LOAD$IO$JOB	Creates an I/O job, loads the job's code, and causes the job's task to run.	Synchronous	2-25
S$OVERLAY	Loads an overlay into memory.	Synchronous	2-32

A$LOAD

The A$LOAD system call loads an object code or data file from secondary storage into memory.

```
CALL RQ$A$LOAD(connection, response$mbox, except$ptr);
```

INPUT PARAMETERS

connection	A TOKEN for a connection to the file that the Loader is to load. The connection must satisfy all of the following requirements: • It must have been created in the calling task's job. • It must be a connection to a named file. • When the file was created by CREATE$FILE or ATTACH$FILE, the specified user object must have had READ access to the file. • It must be closed. If the connection does not satisfy all four of these requirements, the Loader returns an exception code.
response$mbox	A TOKEN for the mailbox to which the Loader sends the Loader Result Segment after the concurrent part of the system call finishes running. The format of the Loader Result Segment is given in the following DESCRIPTION section.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the Loader is to place the condition code generated by the sequential part of the system call.

DESCRIPTION

A$LOAD allows your task to load object code files or data files from secondary storage into main memory. Unlike the A$LOAD$IO$JOB and S$LOAD$IO$JOB system calls, A$LOAD doesn't automatically cause the code to be executed as a task. The calling task must explicitly cause the code to be executed. The following sections explain how to use A$LOAD to load main modules or to load procedures and they give guidelines for calling CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB to run the loaded code.

Using A$LOAD to Load a Main Module

If you are using the A$LOAD system call to load a main module that will run as a task, there are two cases.

1. The usual case is when you are loading PIC or LTL code, or you are loading absolute code generated with the NOINITCODE control of the LOC86 command. In this case, the Loader returns, in the Loader Result Segment, parameters defining the entry point and stack requirements for the loaded code. Your application needs these parameters when invoking the CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system call.

   If the Loader has been configured to load only absolute code, it will not load main modules generated with the NOINITCODE control. In this event, the Loader returns the E$LOADER$SUPPORT condition code. (See Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.)

2. The unusual case is when your object code is absolute code generated without the NOINITCODE control of the LOC86 command. In this case, you must allow the iRMX 86 Nucleus to create a stack for you. To do this, specify 0:0 for the stack pointer parameter of the CREATE$TASK or the CREATE$JOB system call.

   This action causes the Nucleus to create a stack for the loaded code. However, because the loaded code contains a main module, it also contains code that switches the stack register values so the the Nucleus-created stack is ignored. This stack switching allows the loaded code to use the stack allocated by the SEGSIZE control.

   To minimize the amount of memory wasted by stack switching, specify a small stack size (128 decimal bytes) in CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system calls. This stack need not be large because it is used only if the task is interrupted and stack switching occurs.

Stack switching has an undesirable but avoidable side effect. If you use the iRMX 86 Debugger, it will always indicate that the stack for the loaded code has overflowed. The overflow indication is caused by the main module switching stacks, rather than by an actual overflow. This means that you cannot tell whether overflow actually has occurred. To avoid this side effect, write your source code as a procedure or use the LOC86 NOINITCODE control.

Using A$LOAD To Load A Procedure

If you write code as a procedure that you intend to load and run, it can be loaded only by A$LOAD. Although the process of loading a procedure is more restrictive than that of loading a main module, you can avoid the stack-switching side effects described in the previous section.

To successfully load code that is written as a procedure, adhere to the following rules:

- Generate the procedure as absolute code and do not use the NOINITCODE control of the LOC86 command.
- Adhere to the PL/M-86 LARGE model of segmentation. This means that you must either compile the procedure using the LARGE size control, or you must follow the calling conventions of the LARGE model. For information about the PL/M-86 LARGE model of segmentation, refer to the PL/M-86 USER'S GUIDE.
- When invoking the LOC86 command to assign absolute addresses to your object code, use the START control to select one of the PUBLIC symbols in your procedure as an entry point. Also specify SEGSIZE(STACK(0)) to set the stack to zero length. For more information about the START and SEGSIZE controls, refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE.
- When you invoke the CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system call, allow the Operating System to allocate a stack for the new task. Do this by setting the stack pointer parameter to 0:0. Be certain that you specify a stack size parameter that is large enough for the task. For guidelines to determining stack sizes, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.
- When you invoke the CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system call, set the data segment base parameter to 0. The reason for this is that a procedure adhering to the LARGE model of segmentation always initializes its own data segment.

For information about the CREATE$TASK or the CREATE$JOB system calls refer to the iRMX 86 NUCLEUS REFERENCE MANUAL. For information about the CREATE$IO$JOB system call, refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL. For information about the iRMX 86 Debugger, refer to the iRMX 86 DEBUGGER REFERENCE MANUAL.

Asynchronous Behavior

The A$LOAD system call is asynchronous. It allows the calling task to continue running while the loading operation is in progress. When the loading operation is finished, the Loader sends a Loader Result Segment to the mailbox designated by the response$mbox parameter. Refer to Appendix C for an explanation of how asynchronous system calls work.

File Sharing

The Loader does not expect exclusive access to the file. However, other tasks sharing the file are affected by the following:

- The other tasks should not attempt to share the connection passed to the Loader, but instead should obtain their own connections to the file.

- The Loader specifies "share with readers only" when opening the connection, so, during the loading operation, other tasks can access the file only for reading.

Considerations Relating To Code Type

If the file being loaded contains absolute code, the Loader will not create iRMX 86 segments for the code. Rather, it will simply load the program into the memory locations specified for the target file. It is the user's responsibility to prevent code from loading over existing information, including the Operating System code. Refer to the iRMX 86 CONFIGURATION GUIDE to see how to do this by reserving areas of memory.

In contrast, if the file being loaded is position-independent code or load-time locatable code, the Loader will create iRMX 86 segments for containing the loaded program. However, the Loader does not delete these segments; when your task no longer needs the loaded program, your task should delete the segments.

Effects Of Model Of Segmentation

The Loader will return (in the Loader Result Segment) a token for each of the code, data, and stack segments. This is enough segment information for programs compiled as COMPACT, because only one segment of each type will be created. But if the program adheres to the LARGE or MEDIUM model of segmentation, more than one code segment and more than one data segment can be created, although only one token will be returned for each in the Loader Result Segment.

This means that if the code follows the LARGE or MEDIUM model, the calling task cannot know the location of all of the loaded program's code or data segments. Consequently, the calling task cannot delete all of the data or code segments after the program has executed.

You can avoid this problem in either of two ways. Either be certain that the program being loaded adheres to the COMPACT model of segmentation, or use the A$LOAD$IO$JOB or S$LOAD$IO$JOB system calls instead of the A$LOAD system call.

Format Of The A$LOAD Loader Result Segment

The Loader uses memory from the pool of the calling task's job to create the Loader Result Segment for this system call. The calling task should delete the segment after it is no longer needed. Creating multiple segments without deleting them can result in an E$MEM exception code.

The Loader Result Segment has the following form:

```
    STRUCTURE       (except$code         WORD,
                     record$count        WORD,
                     error$rec$type      BYTE,
                     undefined$ref       WORD,
                     init$ip             WORD,
                     code$seg$base       WORD or SELECTOR,
                     stack$offset        WORD,
                     stack$seg$base      WORD or SELECTOR,
                     stack$size          WORD,
                     data$seg$base       WORD or SELECTOR);
```

where:

except$code	A WORD containing the condition code for the concurrent part of the system call. If the code is other than E$OK, some problem occurred during the loading operation.
record$count	A WORD containing the number of records read by the Loader on this invocation of A$LOAD. If the the loading operation terminates prematurely, record$count contains the number of the last record read.
error$rec$type	A BYTE identifying the type of record causing premature termination of the loading operation, except that a value of 0 means no record caused premature termination. Object record types are documented in the Intel publication 8086 RELOCATABLE OBJECT MODULE FORMATS.
undefined$-ref	A WORD specifying whether the Loader found undefined external references while loading the job. An undefined external reference usually results from a linking error. The Loader continues to run even if a target file contains undefined external references.

The value of undefined$ref depends upon your configuration of the Loader. (See Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.)

- If the Loader is configured to load LTL and overlay code, as well as PIC and absolute code, undefined$ref contains the number of undefined external references detected during the loading operation. (Note that undefined$ref equals the number of undefined external references even if the Loader is loading PIC or absolute code.)

- If the Loader is configured to load only absolute code or only PIC or absolute code, the Loader sets undefined$ref to 1 or to 0. It is 1 if the Loader finds undefined external references; otherwise, it is 0.

init$ip

A WORD containing the initial value for the loaded program's instruction pointer (IP register). The calling task can use this information in either of two ways:

- When invoking the CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system call.

- As the destination of a jump within the code segment of the loaded program.

Init$ip is 0 if the file does not specify an initial value for the instruction pointer, as can happen when the file contains no main module.

code$seg$base

A WORD or SELECTOR containing the base address for the code segment with the entry point. The value in code$seg$base can be used with init$ip as a POINTER to the entry point of the loaded program. The Loader places 0 into this field if the loaded program does not contain a main module. If you are using a compiler that supports the data type SELECTOR, code$seg$base should be declared a SELECTOR.

stack$offset

A WORD containing the offset of the bottom of the stack, relative to the beginning of the stack segment. The calling task can use the sum of this value and the stack$size to initialize the stack pointer (SP register).

The Loader sets stack$offset to zero under each of these circumstances:

- The stack actually starts at offset 0.
- There is no main module.
- The loaded code is a main module that dynamically initializes the SP and SS registers.

stack$seg$base	A WORD or SELECTOR containing the base of the stack segment for the loaded program. The calling task can use this value to initialize the stack segment (SP register). Stack$seg$base should be declared a SELECTOR if your compiler supports the SELECTOR data type.

The Loader sets stack$seg$base to 0 under each of these circumstances:

- If there is no main module. (In this case, the target file does not specify a stack base).
- If the loaded code is a main module that dynamically initializes the SP and SS registers.

stack$size	A WORD specifying the number of bytes required for the loaded program's stack. The calling task can initialize the stack pointer (SP register) to the sum of stack$offset and stack$size when invoking the CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system call.

The Loader sets this value to 0 whenever both the stack$offset and stack$seg$base are 0. When all three stack-related parameters are 0 and the target file contains a main module, the loaded code must set the stack pointer (SP register) and stack segment (SS register).

data$seg$base	A WORD or SELECTOR containing the initial base address of the data segment (DS register). If your compiler supports the SELECTOR data type, data$seg$base should be declared a SELECTOR.

The Loader sets this value to 0 under each of these circumstances:

- If the target file contains no main module.
- If the main module dynamically sets the DS register after the program starts running.

CONDITION CODES

The A$LOAD system call can return condition codes at two different times. Codes returned to the calling task immediately after invocation of the system call are sequential condition codes. Codes returned after the concurrent part of the system call has finished running are concurrent condition codes. The following list is divided into two parts -- one for sequential codes and one for concurrent codes:

Sequential Condition Codes

The Loader can return any of the following condition codes to the WORD pointed to by the except$ptr parameter of this system call.

E$OK	No exceptional conditions.
E$BAD$HEADER	The target file does not begin with a valid header record for a loadable object module. Possibly the file is a directory.
E$CHECKSUM	The header record of the target file contains a checksum error.
E$CONN$NOT$OPEN	The Loader opened the connection but some other task closed the connection before the loading operation was begun.
E$CONN$OPEN	The calling task specified a connection that was already open.
E$EXIST	At least one of the following is true: • The connection parameter is not a token for an existing object. • The msg$mbox parameter did not refer to an existing object.
E$FACCESS	The specified connection did not have "read" access to the file.
E$FLUSHING	The device containing the target file is being detached.
E$IO$HARD	A hard I/O error occurred. This means that another try is probably useless.
E$IO$OPRINT	The device containing the target file was off-line. Operator intervention is required.
E$IO$SOFT	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.

E$IO$UNCLASS	An unknown type of I/O error occurred.
E$IO$WRPROT	The volume is write-protected.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • Either the calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations.
E$LOADER$SUPPORT	To load the target file requires capabilities not configured into the Loader. For example, it might be attempting to load PIC when configured to load only absolute code.
E$MEM	The memory available to the calling task's job or the Basic I/O System is not sufficient to complete the call.
E$NOT$FILE$CONN	The calling task specified a connection to a device rather than to a named file.
E$SHARE	The calling task tried to open a connection to a file already being used by some other task, and the file's sharing attribute is not compatible with the open request.
E$SUPPORT	The specified connection was not created by the calling task's job.
E$TYPE	The connection parameter is a token for an object that is not a connection.

Concurrent Condition Codes

After the Loader attempts the loading operation, it returns a condition code in the except$code field of the Loader Result Segment. The Loader can return the following condition codes in this manner.

E$OK	No exceptional conditions.
E$BAD$GROUP	The target file contains an invalid group definition record.
E$BAD$SEGMENT	The target file contains an invalid segment definition record.
E$CHECKSUM	At least one record of the target file contains a checksum error.

E$EOF	The call encountered an unexpected end-of-file.
E$EXIST	At least one of the following is true: • The mailbox specified in the response$mbox parameter was deleted before the loading operation was completed. • The device containing the file to be loaded was detached before the loading operation was completed.
E$FIXUP	The target file contains an invalid fixup record.
E$FLUSHING	The device containing the target file is being detached.
E$IO$HARD	A hard I/O error occurred. This means that another try is probably useless.
E$IO$OPRINT	The device containing the target file was off-line. Operator intervention is required.
E$IO$SOFT	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred.
E$IO$WRPROT	The volume is write-protected.
E$LIMIT	The calling task's job has already reached its object limit.
E$NO$LOADER$MEM	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Loader to run.
E$NO$MEM	The Loader attempted to load PIC or LTL groups or segments, but the memory pool of the calling task's job does not currently contain a block of memory large enough to accommodate these groups or segments.
E$NOSTART	The target file does not specify the entry point for the program being loaded.
E$PARAM	The target file has a stack smaller than 16 bytes.
E$REC$FORMAT	At least one record in the target file contains a format error.

E$REC$LENGTH	The target file contains a record longer than the Loader's internal buffer. The Loader's buffer length is specified during the configuration of the Loader. See Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.
E$REC$TYPE	At least one of the following is true: • At least one record in the target file is of a type that the Loader cannot process. • The Loader encountered records in a sequence that it cannot process.
E$SEG$BOUNDS	The Loader created a segment into which to load code. One of the data records specified a load address outside of that segment.

A$LOAD$IO$JOB

The A$LOAD$IO$JOB system call reads the header record of an executable file in secondary storage and creates an I/O job. The job's initial task then performs the concurrent part of the call, which is the loading of the remainder of the file.

```
    job = RQ$A$LOAD$IO$JOB(connection, pool$lower$bound, pool$upper$bound,
                           except$handler, job$flags, task$priority,
                           task$flags, msg$mbox, except$ptr);
```

INPUT PARAMETERS

connection	A TOKEN for a connection to the file that the Loader will load. The connection must be a connection to a named file. Also, the connection must be closed, the user object specified when the connection was created must have had READ access, and the connection must have been created in the calling task's job. The Loader opens the connection for sharing with readers only, so, during the loading operation, other tasks may access the file only for reading.
pool$lower$-bound	A WORD containing a value the Loader uses to compute the pool size for the new I/O job. See the DESCRIPTION section for details.
pool$upper$-bound	A WORD containing a value the Loader uses to compute the pool size for the new I/O job. See the DESCRIPTION section for details.
except$handler	A POINTER to a structure of the following form:

```
STRUCTURE(
    exception$handler$offset      WORD,
    exception$handler$base        WORD or SELECTOR,
    exception$mode                BYTE)
```

The Loader expects you to designate one exception handler to be used both for the new task and for the new job's default exception handler. If you want to designate the system default exception handler, you can do so by setting exception$handler$base to zero. If you set the base to any other value, then the Loader assumes that the first two words of this structure point to the first instruction of your exception handler.

Exception$handler$base should be declared a SELECTOR if the compiler you are using supports the SELECTOR data type.

Set the exception$mode to specify when control is to pass to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Control never passes to handler
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

For more information regarding exception handlers and the exception mode, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

job$flags A WORD specifying whether the Nucleus is to check the validity of objects used as parameters in system calls. Setting bit 1 (where bit 0 is the low-order bit) to 0 specifies that the Nucleus is to check the validity of objects. All bits other than bit 1 must be set to 0.

task$priority A BYTE which,

- if equal to 0, indicates that the new job's initial task is to have a priority equal to the maximum priority of the initial job of the Extended I/O System.

- if not equal to 0, contains the priority of the initial task of the new job. If this priority is higher (numerically lower) than the maximum priority of the initial job of the Extended I/O System, an E$PARAM error occurs.

task$flags A WORD indicating whether the initial task uses floating-point instructions, and whether to start the task immediately.

Set bit 0 (the low-order bit) to 1 if the task uses floating-point instructions; otherwise set it to 0.

Bit 1 indicates whether the initial task in the job should run immediately, or whether it should be suspended until a START$IO$JOB system call is issued to start it. Set it to 0 if the task is to be made ready immediately; set it to 1 if the task is to be suspended.

Set bits 2 through 15 to 0.

msg$mbox	A TOKEN for a mailbox that serves two purposes. The first purpose is to receive the Loader Result Segment after the loading operation is completed. The format of the Loader Result Segment is provided later in this description. The second purpose is to receive an exit message from the newly created I/O job. The description of the CREATE$IO$JOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL shows the format of an exit message.

OUTPUT PARAMETERS

except$ptr	A POINTER to a WORD where the Loader is to place the condition code generated by the sequential part of the system call.
job	A TOKEN, returned by the Loader, for the newly created I/O job. This token is valid only if the Loader returns an E$OK condition code to the WORD pointed to by the except$ptr parameter.

DESCRIPTION

This system call operates in two phases. The first phase occurs during the sequential part of this system call. (Refer to Appendix C for a discussion of the sequential and concurrent parts of an asynchronous system call.) During this first phase, the Loader does the following:

- Checks the validity of the header record of the target file.
- Creates an I/O job. This I/O job is a child of the calling task's job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a definition of I/O jobs.)
- Returns a condition code reflecting the success or failure of the first phase. The Loader places this condition code in the WORD pointed to by the except$ptr parameter.

The second phase occurs during the concurrent part of the system call. This part runs as the initial task in the new job and does the following:

- Loads the file designated by the connection parameter.
- Creates the task that will execute the loaded code. If there are no errors while the file is being loaded and if bit 1 of the task$flags parameter is 0, the concurrent part makes the task in the new job ready to run.

- Sends a Loader Result Segment to the mailbox specified by the msg$mbox parameter. One element in this segment is a condition code indicating the success or failure of the second phase.

- Deletes itself.

Restriction

This system call should be invoked only by tasks running within I/O jobs. Failure to heed this restriction causes a sequential exception condition.

Pool Size For The New Job

The Loader uses the following information to compute the size of the memory pool for the new I/O job:

- The pool$lower$bound parameter, as a number of 16-byte paragraphs.

- The pool$upper$bound parameter, as a number of 16-byte paragraphs.

- A Loader configuration parameter specifying the default dynamic memory requirements. (Refer to Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.)

- Memory requirements specified in the target file.

The Loader gives you three options for setting the size of the I/O job's memory pool:

1. You can set both pool$lower$bound and pool$upper$bound to 0. If you do this, the Loader decides how large a 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 or both of the bound parameters to override the Loader's decision on pool size. If the Loader's decision lies outside the bound(s) that you specify, the Loader adjusts its decision so that it complies with your bounds.

3. If you set pool$upper$bound to OFFFFH, the Loader allows the new I/O job to borrow memory from the calling task's job. The initial size of the memory pool is based on the pool$lower$bound parameter.

If you select Option 1 or 2, the Loader creates an I/O job with the minimum pool size equal to the maximum pool size. This means that the new I/O job will not be able to borrow memory from the calling task's job. If you want the I/O job to be able to borrow memory, select Option 3.

This system call is asynchronous. It allows the calling task to continue running while the loading operation is in progress. When the loading operation is finished the Loader sends a Loader Result Segment to the mailbox designated by the msg$mbox parameter. Refer to Appendix C for a detailed description of asynchronous system call behavior.

Format Of The Loader Result Segment

The Loader Result Segment has the form described below. This structure is deliberately compatible with the structure of the message returned when an I/O job exits. (See the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a description of exit messages.)

```
STRUCTURE     (termination$code       WORD,
                  except$code           WORD,
                  job$token             TOKEN,
                  return$data$len       BYTE,
                  record$count          WORD,
                  error$rec$type        BYTE,
                  undefined$ref         WORD,
                  mem$requested         WORD,
                  mem$received          WORD);
```

where:

termination$code	A WORD indicating the success or failure of the loading operation. • A value of 100H indicates that the loading operation succeeded. • A value of 2 indicates that the loading operation failed. In this case, your system should delete the newly created I/O job; the Loader doesn't do so.
except$code	A WORD containing the concurrent condition code. Codes and interpretations follow this description.
job$token	A TOKEN for the newly created I/O job.
return$data$len	A BYTE that is always set to 9.

record$count	A WORD containing the number of records read by the Loader. If the loading operation terminates prematurely, this value indicates the last record read.
error$rec$type	A BYTE identifying the reason the loading operation terminated. • A value of 0 means that no record caused termination. • A non-0 value is the type of the record that caused premature termination. Object record types are documented in the Intel publication 8086 RELOCATABLE OBJECT MODULE FORMATS.
undefined$-ref	This value tells whether the Loader found undefined external references while loading the job. An undefined external reference usually results from a linking error. The Loader continues to run even if an target file contains undefined external references. The value of undefined$ref depends upon the configuration of the Loader. (See Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.) • If the Loader is configured to load LTL code, as well as PIC and absolute code, undefined$ref contains the number of undefined external references the Loader detected during the loading operation. (Note that undefined$ref equals the number of undefined external references even if the Loader is loading PIC or absolute code.) • If the Loader is configured to load only PIC or absolute code or only absolute code, the Loader sets undefined$ref to 1 or to 0. It is 1 if the Loader found undefined external references; otherwise, it is 0.
mem$requested	A WORD indicating the number of 16-byte paragraphs the target file requested for the new job, including the memory needed for all segments and that needed for the job's memory pool.
mem$received	A WORD indicating the number of 16-byte paragraphs actually allocated to the new job.

CONDITION CODES

This system call can return condition codes at two different times. Codes returned to the calling task immediately after the invocation of the system call are considered sequential condition codes. Codes returned after the concurrent part of the system call has finished running are considered concurrent condition codes. The following list is divided into two parts -- one for sequential codes and one for concurrent codes.

Sequential Condition Codes

The Loader returns one of the following condition codes to the WORD pointed to by the except$ptr parameter:

E$OK	No exceptional conditions.
E$BAD$HEADER	The target file does not begin with a valid header record for a loadable object module. Possibly the file is a directory.
E$CHECKSUM	The header record of the target file contains a checksum error.
E$CONN$NOT$OPEN	The Loader opened the connection, but some other task closed the connection before the loading operation was begun.
E$CONN$OPEN	The specified connection was already open.
E$CONTEXT	The calling task's job is not an I/O job.
E$EXIST	At least one of the following is true: • The connection parameter is not a token for an existing object. • The calling task's job has no global job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a definition of global job. • The msg$mbox parameter does not refer to an existing object.
E$FACCESS	The specified connection does not have "read" access to the file.
E$FLUSHING	The device containing the target file is being detached.
E$IO$HARD	A hard I/O error occurred. This means that another try is probably useless.

E$IO$OPRINT	The device containing the target file is off-line. Operator intervention is required.
E$IO$SOFT	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred.
E$IO$WRPROT	The volume is write-protected.
E$JOB$PARAM	The pool$upper$bound parameter is both non-zero and smaller than the pool$lower$bound parameter.
E$JOB$SIZE	The pool$upper$bound parameter is non-0 and too small for the target file.
E$LOADER$SUPPORT	The target file requires capabilities not configured into the Loader. For example, the loader might be attempting to load PIC code when configured to load only absolute code.
E$MEM	The memory available to the calling task's job or the Basic I/O System is not sufficient to complete the call.
E$NO$LOADER$MEM	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Loader to run.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$NOT$FILE$CONN	The specified connection is to a device rather than to a named file.
E$PARAM	The value of the except$mode field within the except$handler structure lies outside the range 0 through 3.
E$SHARE	The calling task tried to open a connection to a file already being used by some other task, and the file's sharing attribute is not compatible with the open request.
E$SUPPORT	The specified connection was not created in this job.
E$TIME	The calling task's job is not an I/O job.
E$TYPE	The connection parameter is a token for an object that is not a connection.

Concurrent Condition Codes

After the Loader attempts the loading operation, it returns a condition code in the except$code field of the Loader Result Segment. The Loader can return the following condition codes in this manner:

E$OK	No exceptional conditions.
E$BAD$GROUP	The target file contains an invalid group definition record.
E$BAD$SEGMENT	The target file contains an invalid segment definition record.
E$CHECKSUM	At least one record of the target file contains a checksum error.
E$EOF	The call encountered an unexpected end-of-file.
E$EXIST	At least one of the following is true: • The mailbox specified in the msg$mbox parameter was deleted before the loading operation was completed. • The device containing the target file was detached before the loading operation was completed.
E$FACCESS	The default user of the newly created I/O job does not have "read" access to the target file.
E$FIXUP	The target file contains an invalid fixup record.
E$FLUSHING	The device containing the target file is being detached.
E$IO$HARD	A hard I/O error occurred. This means that another try is probably useless.
E$IO$OPRINT	The device containing the target file is off-line. Operator intervention is required.
E$IO$SOFT	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred.
E$IO$WRPROT	The volume is write-protected.
E$LIMIT	At least one of the following is true:

	• The task$priority parameter is higher (numerically lower) than the newly-created I/O job's maximum priority. This maximum priority is specified during the configuration of the Extended I/O System (if the job is a descendant of the Extended I/O System) or during configuration of the Human Interface (if the job is a descendant of the Human Interface).
	• Either the newly created I/O job, or its default user, is already involved in 255 (decimal) I/O operations.
E$NO$LOADER$MEM	There is not sufficient memory available to the newly created I/O job or the Basic I/O System to allow the Loader to run.
E$NO$MEM	The Loader is attempting to load PIC or LTL groups or segments, but the memory pool of the newly created I/O job does not currently contain a block of memory large enough to accommodate these groups or segments.
E$NOSTART	The target file does not specify the entry point for the program being loaded.
E$PARAM	The target file has a stack smaller than 16 bytes.
E$REC$FORMAT	At least one record in the target file contains a format error.
E$REC$LENGTH	The target file contains a record longer than the Loader's internal buffer. The internal buffer length is specified during the configuration of the Loader. Refer to Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.
E$REC$TYPE	At least one of the following is true: • At least one record in the target file is of a type that the Loader cannot process. • The Loader encountered records in a sequence that it cannot process.
E$SEG$BOUNDS	The Loader created a segment into which to load code. One of the data records specified a load address outside of the new segment.

S$LOAD$IO$JOB

The S$LOAD$IO$JOB system call creates an I/O job containing the Loader task, which loads the code for the user task from secondary storage.

```
    job = RQ$S$LOAD$IO$JOB(path$ptr, pool$lower$bound, pool$upper$bound,
                           except$handler, job$flags, task$priority,
                           task$flags, msg$mbox, except$ptr);
```

INPUT PARAMETERS

path$ptr	A POINTER to a STRING containing a path name for the named file with the object code to be loaded. The path name must conform to the Extended I/O System path syntax for named files. If you are not familiar with iRMX 86 path syntax, refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.
pool$lower$-bound	A WORD containing a value that the Loader uses to compute the pool size for the new I/O job. See the DESCRIPTION section for details.
pool$upper$-bound	A WORD containing a value that the Loader uses to compute the pool size for the new I/O job. See the DESCRIPTION section for details.
except$handler	A POINTER to a structure of the following form:

```
STRUCTURE    (exception$handler$offset      WORD,
              exception$handler$base        WORD or SELECTOR,
              exception$mode                BYTE)
```

The Loader expects you to designate an exception handler to be used both for the new task and for the new job's default exception handler. If you want to designate the system default exception handler, do so by setting exception$handler$base to 0. If you set the base to any other value, then the Loader assumes that the first two words of this structure point to the first instruction of your exception handler.

Exception$handler$base should be declared as a SELECTOR if the compiler you are using supports the SELECTOR data type.

except$handler (continued)

Set the exception$mode to tell the Loader when to pass control to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Control never passes to handler
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

For more information regarding exception handlers and the exception mode, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

job$flags

A WORD specifying whether the Nucleus is to check the validity of objects used as parameters in system calls. Setting bit 1 (where bit 0 is the low-order bit) to 0 specifies that the Nucleus is to check the validity of objects. All bits other than bit 1 must be set to 0.

task$priority

A BYTE which,

- if equal to 0, indicates that the new job's initial task is to have a priority equal to the the maximum priority of the initial job of the Extended I/O System.
- if not equal to 0, contains the priority of the initial task of the new job. If this priority is higher (numerically lower) than the maximum priority of the initial job of the Extended I/O System, an E$PARAM error occurs.

task$flags

A WORD indicating whether the initial task uses floating-point instructions, and whether to start the task immediately.

Set bit 0 (the low-order bit) to 1 if the task uses floating-point instructions; otherwise set it to 0.

Bit 1 indicates whether the initial task in the job should run immediately, or whether it should be suspended until a START$IO$JOB system call is issued to start it. Set bit 1 to 0 if the task is to be made ready immediately; set it to 1 if the task is to be suspended.

Set bits 2 through 15 to 0.

msg$mbox	A TOKEN for a mailbox that receives an exit message from the newly created I/O job. The description of the CREATE$IO$JOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL documents the format of an exit message.

OUTPUT PARAMETERS

except$ptr	A POINTER to a WORD where the Loader is to place a condition code.
job	A TOKEN, returned by the Loader, for the newly created I/O job. This token is valid only if the Loader returns an E$OK condition code to the WORD specified by the except$ptr parameter.

DESCRIPTION

This system call performs the same function as A$LOAD$IO$JOB. The only difference between the calls is that S$LOAD$IO$JOB is synchronous. That is, the calling task resumes running only after the call has completed its attempt to create an I/O job and a user task in that job.

The Loader does not necessarily have exclusive access to the file being loaded. During the loading operation, however, if other tasks are also using the file, they may access the file only for reading.

NOTE

This system call should be invoked only by tasks running within I/O jobs. Failure to heed this restriction causes the Loader to return an E$CONTEXT exception code.

Pool Size For The New Job

The Loader uses the following information to compute the size of the memory pool for the new I/O job:

- The pool$lower$bound parameter, as a number of 16-byte paragraphs.
- The pool$upper$bound parameter, as a number of 16-byte paragraphs.
- A Loader configuration parameter specifying the default dynamic memory requirements. (Refer to Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.)
- Memory requirements specified in the target file.

The Loader gives you three options for setting the size of the I/O job's memory pool:

1. You can set both pool$lower$bound and pool$upper$bound to zero. If you do this, the Loader decides how large a 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 or both of the bound parameters to override the Loader's decision on pool size. If the Loader's decision lies outside the bound(s) that you specify, the Loader adjusts it to comply with your bounds.

3. If you set pool$upper$bound to OFFFFH, the Loader allows the new I/O job to borrow memory from the calling task's job. The initial size of the memory pool is equal to pool$lower$bound.

If you select Option 1 or 2, the Loader creates an I/O job with the minimum pool size equal to the maximum pool size. This means that the new I/O job will not be able to borrow memory from the calling task's job. If you want the I/O job to be able to borrow memory, select Option 3.

CONDITION CODES

The Loader returns one of the following condition codes to the WORD specified by the except$ptr parameter of this system call:

E$OK	No exceptional conditions.
E$BAD$GROUP	The target file contains an invalid group definition record.
E$BAD$HEADER	The target file does not begin with a valid header record for a loadable object module.
E$BAD$SEGMENT	The target file contains an invalid segment definition record.
E$CHECKSUM	At least one record in the target file contains a checksum error.
E$CONTEXT	The calling task's job is not an I/O job.
E$EOF	The call encountered an unexpected end-of-file.

E$EXIST	At least one of the following is true: • The msg$mbox parameter is not a token for an existing object. • The calling task's job has no global job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a definition of global job.) • The device containing the target file was detached.
E$FACCESS	The default user object for the new I/O job does not have "read" access to the specified file.
E$FIXUP	The target file contains an invalid fixup record.
E$FNEXIST	The specified target file, or some file in the specified path, does not exist or is marked for deletion.
E$FLUSHING	The device containing the target file is being detached.
E$INVALID$FNODE	The fnode for the specified file is invalid, so the file must be deleted.
E$IO$HARD	A hard I/O error occurred. This means that another try is probably useless.
E$IO$JOB	The calling task's job is not an I/O job.
E$IO$OPRINT	The device containing the target file is off-line. Operator intervention is required.
E$IO$SOFT	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred.
E$IO$WRPROT	The volume is write-protected.
E$JOB$PARAM	The pool$upper$bound parameter is nonzero and smaller than the pool$lower$bound parameter.
E$JOB$SIZE	The pool$upper$bound parameter is nonzero and too small for the target file.

E$LIMIT	At least one of the following is true: • The task$priority parameter is higher (numerically lower) than the newly-created I/O job's maximum priority. This maximum priority is specified during the configuration of the Extended I/O System (if the job is a descendant of the Extended I/O System) or of the Human Interface (if the job is a descendant of the Human Interface). • Either the newly created I/O job or its default user object is already involved in 255 (decimal) I/O operations.
E$LOADER$SUPPORT	The target file requires capabilities not configured into the Loader. For example, it might be attempting to load PIC when configured to load only absolute code.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$NO$LOADER$MEM	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Loader to run.
E$NOMEM	The target file contains either PIC segments or groups, or LTL segments or groups. In any case, the memory pool of the new I/O job does not have a block of memory large enough to allow the Loader to load these records.
E$NOSTART	The target file does not specify the entry point for the program being loaded.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$PARAM	At least one of the following is true: • The value of the except$mode field within the except$handler structure lies outside the range 0 through 3. • The target file requested a stack smaller than 16 bytes.
E$PATHNAME$-SYNTAX	The specified pathname contains one or more invalid characters.
E$REC$FORMAT	At least one record in the target file contains a format error.

E$REC$LENGTH	The target file contains a record longer than the Loader's internal buffer. The Loader's buffer length is specified during the configuration of the Loader. (See Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.)
E$REC$TYPE	At least one of the following is true: • At least one record in the target file is of a type that the Loader cannot process. • The Loader encountered records in a sequence that it cannot process.
E$SEG$BOUNDS	The Loader created a segment into which to load code. One of the data records specified a load address outside of the new segment.

S$OVERLAY

In programs with overlays, the root module of the program calls S$OVERLAY to load overlay modules.

```
    CALL RQ$S$OVERLAY(name$ptr, except$ptr);
```

INPUT PARAMETER

name$ptr	A POINTER to a STRING containing the name of an overlay. The overlay name should have only upper-case letters, both in this string and when you specify the name in the LINK86 OVERLAY control. For information about LINK86, refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Loader will place a condition code.

DESCRIPTION

Root modules issue this system call when they want to load an overlay module. Chapter 1 describes overlays.

Synchronous Behavior

This system call is synchronous. The calling task resumes running only after the system call has completed its attempt to load the overlay.

File Sharing

The Loader does not expect exclusive access to the file containing the overlay module. However, while the overlay is being loaded, if other tasks are also using the file, they can access the file only for reading.

CONDITION CODES

The Loader returns one of the following condition codes to the calling task:

E$OK	No exceptional conditions.
E$CHECKSUM	At least one record in the target overlay contains a checksum error.
E$EOF	The call encountered an unexpected end-of-file.
E$EXIST	The specified device does not exist.
E$FIXUP	The target file contains an invalid fixup record.
E$FLUSHING	The device containing the target file is being detached.
E$IO$HARD	A hard I/O error occurred. This means that another try is probably useless.
E$IO$OPRINT	The device containing the target overlay is off-line. Operator intervention is required.
E$IO$SOFT	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred.
E$IO$WRPROT	The volume is write-protected.
E$LIMIT	Either the calling task's job, or its default user object, is already involved in 255 (decimal) I/O operations.
E$NOMEM	The overlay module contains either PIC segments or groups, or LTL segments or groups. In any case, the memory pool of the new I/O job does not have a block of memory large enough to allow the Loader to load the overlay module.
E$NOT$CONFIGURED	This system call is not part of the present configuration.
E$REC$FORMAT	At least one record in the target overlay contains a format error.
E$REC$LENGTH	The target overlay contains a record longer than the Loader's maximum record length. The Loader's maximum record length is a parameter specified during the configuration of the Loader.

E$REC$TYPE	At least one of the following is true: • At least one record in the target overlay is of a type that the Loader cannot process. • The Loader encountered records in a sequence that it cannot process.
E$OVERLAY	The overlay name indicated by the name$ptr parameter does not match any overlay module name, as specified with the OVERLAY control of the LINK86 command.
E$SEG$BOUNDS	The Loader created a segment into which to load code. One of the data records specified a load address outside of the new segment.

***

CHAPTER 3 CONFIGURATION OF THE APPLICATION LOADER

The Application Loader 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, the iRMX 86 manual set provides three kinds of information:

- A list of configurable options.
- Detailed information about the options.
- Procedures to allow you to specify your choices.

The sections that follow describe the configurable options. To obtain the second and third categories of information, refer to the iRMX 86 CONFIGURATION GUIDE.

TYPES OF JOB-LOADING SYSTEM CALLS

You can select the set of job-loading system calls in your configuration of the Loader. You have these options:

- A$LOAD, which you can choose if you do not intend to load any IO jobs.
- A$LOAD and A$LOAD$IO$JOB, if you do intend to load IO jobs, and if you intend to use only asynchronous loading operations.
- A$LOAD, A$LOAD$IO$JOB, and S$LOAD$IO$JOB, if you want all three options.

LOADER IN ROM

If you intend to place the Loader in ROM, you specify this when you configure your system. If the Loader is not in ROM, it will itself have to be loaded into RAM memory.

TYPE OF CODE TO BE LOADED

You can select the type of code that the Loader can load. The options are:

- Absolute code only
- Position-independent code and absolute code
- Load-time locatable code, absolute code, and position-independent code
- Overlays, as well as absolute, position-independent, and load-time-locatable code

DEFAULT MEMORY POOL SIZE

You must specify the default size of the memory pool for jobs that are created by the A$LOAD$IO$JOB and S$LOAD$IO$JOB system calls. This value can be over-ridden by specifying the memory pool size when using LINK86.

SIZE OF APPLICATION LOADER BUFFERS

You can specify the size of two buffers that the Loader uses to load your programs. The first is called the Read Buffer, and the second is called the Internal Buffer.

***

APPENDIX A DATA TYPES

The following data types are recognized by the iRMX 86 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 address of a (64K-byte processor) segment and an offset in the segment. The offset is in the word having the lower address.
OFFSET	A word whose value represents the distance from the base address of a segment.
SELECTOR	The base address of a segment.
TOKEN	A word or selector whose value identifies an object. A token can be declared literally a WORD or a SELECTOR depending on your needs.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes that follow it in the string.
DWORD	A 4-byte unsigned binary number.

***

APPENDIX B CONDITION CODES

The iRMX 86 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 distinguishing feature between the two kinds of codes is the method that the Loader uses to return the code to the calling task. For a discussion of the difference between these kinds of codes, refer to Appendix C.

The meaning of a specific condition code depends upon the system call that returns the code. For this reason, this appendix does not list interpretations. Refer to Chapter 2 for an interpretation of the codes.

The purpose of this appendix is to provide you with the numeric value associated with each condition code the Loader can return. To use the condition code values in a symbolic manner, you can assign (using the PL/M-86 LITERALLY statement) a meaningful name to each of the codes.

The following list correlates the name of a condition code with the value returned by the Extended I/O System. The list is divided into three parts: one for the normal condition code, one for exception codes indicating a programming error, and one for exception codes indicate an environmental problem. No distinction is drawn between sequential and concurrent errors because most of the codes can be returned as either.

Be aware that this list covers only the condition codes returned by the system calls of the Loader. Additional condition codes can be found in the appendices of one or more of the following manuals:

- iRMX 86 NUCLEUS REFERENCE MANUAL
- iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL
- iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL

NORMAL CONDITION CODE

NAME OF CONDITION	HEXADECIMAL VALUE
E$OK	OH

CONDITION CODES

PROGRAMMER ERROR CODES

NAME OF CONDITION	HEXADECIMAL VALUE
E$JOB$PARAM	8060H

ENVIRONMENTAL PROBLEM CODES

NAME OF CONDITION	HEXADECIMAL VALUE
E$NOT$CONFIGURED	8H
E$IO$JOB	47H
E$IO$UNCLASS	50H
E$IO$SOFT	51H
E$IO$HARD	52H
E$IO$PRINT	53H
E$IO$WRPROT	54H
E$ABS$ADDRESS	60H
E$BAD$GROUP	61H
E$BAD$HEADER	62H
E$BAD$SEGDEF	63H
E$CHECKSUM	64H
E$EOF	65H
E$FIXUP	66H
E$JOB$SIZE	6DH
E$LOADER$SUPPORT	6FH
E$NO$LOADER$MEM	67H
E$NO$MEM	68H
E$NO$START	6CH
E$OVERLAY	6EH
E$REC$FORMAT	69H
E$REC$LENGTH	6AH
E$REC$TYPE	6BH
E$SEG$BOUNDS	70H

***

APPENDIX C ASYNCHRONOUS SYSTEM CALLS

The iRMX 86 Application Loader provides two types of system calls: synchronous and asynchronous. Synchronous calls return control to the calling task after all operations are completed, either successfully or unsuccessfully. But asynchronous calls are more complex. This Appendix describes the operation of iRMX 86 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 C-1 to see how the parts relate.

- the sequential part

  The sequential part behaves in much the same way as the fully synchronous system calls. Its purpose is to verify parameters, check conditions, and prepare the concurrent part of the system call. Also, it returns a condition code. The sequential part then returns control to your application.

- the concurrent part

  The concurrent part runs as an iRMX 86 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 86 Operating System gives it control of the processor. The concurrent part also returns a condition code.

The reason for splitting the asynchronous calls into two parts is 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 Loader allows your application to run while the Loader 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 must load a program that is 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.

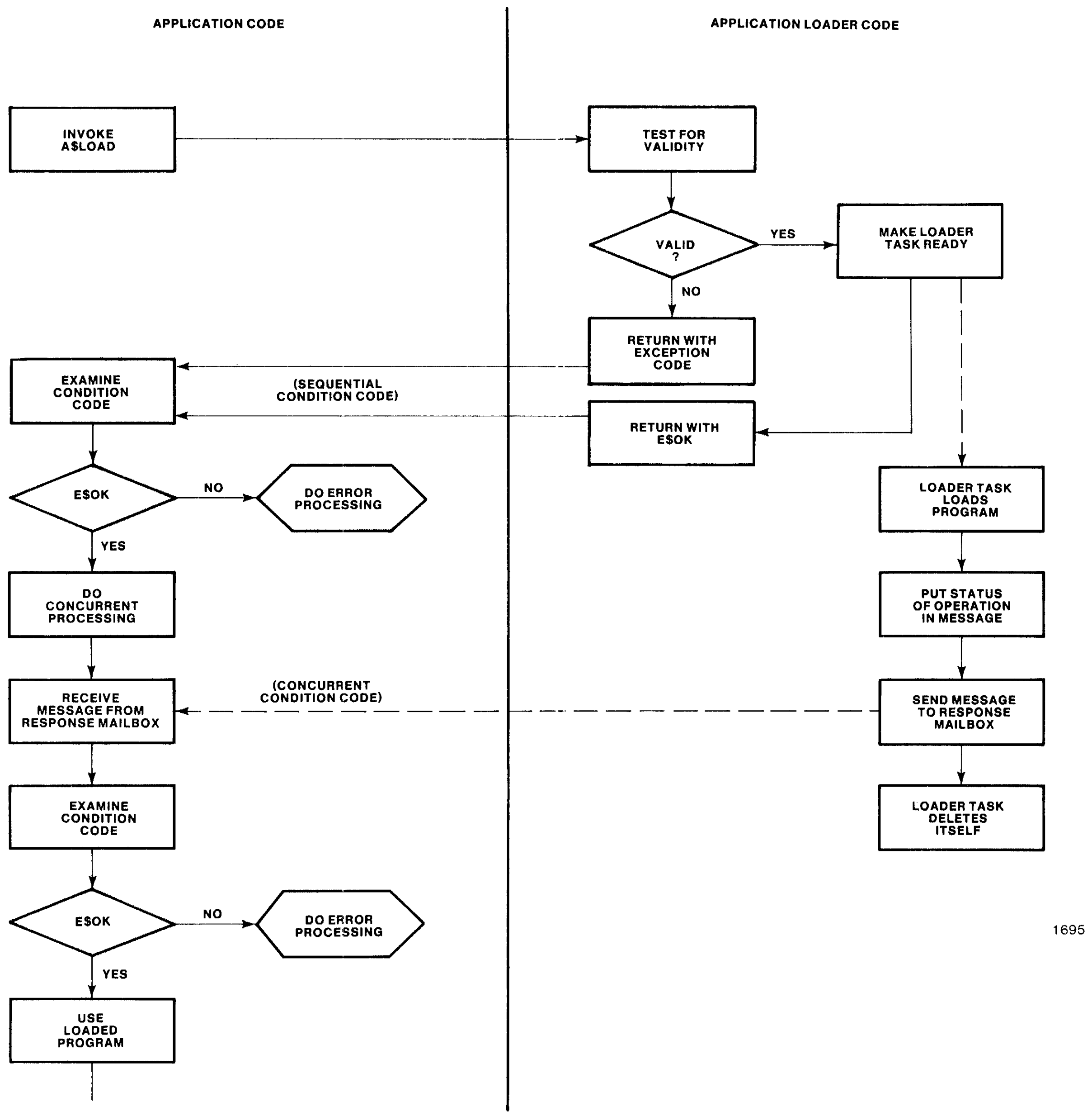


Figure C-1. Behavior Of An Asynchronous System Call

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. It is at this point that 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.

   On the other hand, 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, we will assume that the sequential part of the system call returned an E$OK sequential condition code.

5. Your application now may use the loaded program. But first, your application 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.

   By using the RECEIVE$MESSAGE system call, the application obtains a Loader Result Segment containing a condition code for the concurrent part of the A$LOAD system call. If this condition code is E$OK, then the loading operation was successful, and the application can use the loaded program. On the other hand, if the code is not E$OK, the application should analyze the code and attempt to determine why the loading operation was not successful.

In the foregoing example, we 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 86 asynchronous calls:

- All of the asynchronous system calls consist of two parts -- one sequential and one concurrent. The Loader will activate 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. There can be no message 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 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 Chapter 2 of this manual.

- Whenever 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.

***

INDEX

Primary references are underscored.

***

iRMX™ 86 HUMAN INTERFACE REFERENCE MANUAL

CONTENTS

PAGE

PAGE

CONTENTS (continued)

***

CHAPTER 1 OVERVIEW

The iRMX 86 Human Interface is a layer of the Operating System that allows console operators to load and execute program files (also called commands) from terminals. When the Human Interface begins running, it:

- Creates an iRMX 86 job for each terminal configured in the Human Interface. This job (also called the interactive job) furnishes the application environment; all commands entered by the operator run as offspring jobs of the operator's interactive job.

- Assigns an area of main memory for the operator (this occurs as part of creating the interactive job). Any commands that the operator runs use this area of memory.

- Starts an initial program (this also occurs as part of creating the interactive job). The initial program is the operator's interface to the Operating System. It is 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 invokes the commands based on that terminal input. You can also supply your own initial programs. In fact, there can be a separate initial program for each terminal, if necessary.

When an operator enters information at a Human Interface terminal, the operator communicates with the initial program. With the standard initial program, the operator invokes a command by specifying the pathname of the file that contains the command (and optionally specifying parameters). The initial program reads the information from the terminal and invokes Human Interface system calls to load the command into main memory from secondary storage, create an iRMX 86 job for the command (as an offspring of the operator's interactive job), and begin command execution.

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 standard command line interpreter (CLI).

- Multi-access support.

- Support for wild-card pathnames.

This chapter provides an overview of these features.

RESIDENT HUMAN INTERFACE COMMANDS

In addition to the code for the resident Human Interface, Intel has written a variety of commands which you can use with any application system that includes the Human Interface. Included are:

- 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, SUBMIT, and others)

The iRMX 86 OPERATOR'S MANUAL contains complete descriptions of all commands supplied with the Human Interface.

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 calls

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 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 provides a further discussion of program control.

STANDARD INITIAL PROGRAM

As stated previously, when an operator activates a terminal, 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 iRMX 86 CONFIGURATION GUIDE.

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). The function of the Human Interface CLI is to read input from the terminal and invoke commands based on that input. This CLI (or a user-supplied CLI) is required to allow an operator to invoke commands.

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 user IDs. This support is called multi-access support.

With multi-access support, multiple operators can communicate with the Operating System. The Human Interface assigns each operator a unique identification, called a user ID, and 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.

To run a multi-access Human Interface, a privileged operator (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. This process is described in the iRMX 86 CONFIGURATION GUIDE.

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 difficulty 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.

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 zero characters)

The iRMX 86 OPERATOR'S MANUAL 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.

***

CHAPTER 2
SUPPORTING MULTIPLE TERMINALS

The iRMX 86 Operating System provides two ways for you to implement multiple-terminal support on your application system. You can:

- Write application tasks that use the system calls of the Basic and Extended I/O Systems to communicate directly with multiple terminals.

- Use the multi-access Human Interface.

This chapter discusses both methods.

COMMUNICATING WITH TERMINALS VIA THE BASIC AND EXTENDED I/O SYSTEMS

One method of providing multiple terminal support is to omit the Human Interface from your system, write your own application programs that access the terminals directly, and configure these programs as tasks in the Operating System. The Basic I/O System provides device drivers that allow tasks to communicate with multiple terminals. Therefore, if your system contains the necessary hardware, your application tasks can use Basic and Extended I/O System calls to communicate with each terminal in your system.

If you communicate with the terminals directly, without using the Human Interface, you can tailor your terminal interface to meet your exact needs. This might result in smaller, faster code than the Human Interface (but at the expense of an increased program development effort). This method requires you to write a great deal of code that the Human Interface already supplies.

If you plan to use this method of providing multiple terminal support, none of the information contained in this manual applies to you. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL and the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about the system calls you can use to communicate with terminals.

USING THE MULTI-ACCESS HUMAN INTERFACE

The other method of providing multiple-terminal support is to use the multi-access support provided by the Human Interface. The multi-access support includes code required to communicate with multiple terminals.

It uses the same Basic and Extended I/O System calls that you would have to use if you implemented the method described in the previous section. However, the multi-access Human Interface also provides high-level support for this communication. For example, from a terminal in a multi-access system, an operator can execute commands, run development programs (like editors, compilers, and so on), and run other application programs. If you decide to use the multi-access support features of the Human Interface, you can still tailor your system to meet your individual needs. An important way of doing this is by selecting, for each operator, the initial program that runs when that operator accesses the Human Interface. There are two choices: the initial program supplied with the Human Interface (the standard CLI) or initial programs that you write. The user description files maintained by the system manager identify this choice to the Human Interface (refer to the iRMX 86 CONFIGURATION GUIDE for more information). By selecting the initial program, you can greatly influence the operator's interface to the Human Interface.

STANDARD INITIAL PROGRAM

The Human Interface supplies a command line interpreter (CLI) as the standard initial program. During initialization, the Human Interface CLI performs the following operations:

- Displays a sign-on message.
- Creates an iRMX 86 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 Human Interface CLI performs the following operations:

- Displays the Human Interface prompt (-) and reads input from the terminal (using the Human Interface system call C$SEND$CO$RESPONSE).
- Places the information it reads into the command connection (using the Human Interface system call C$SEND$COMMAND). After receiving a complete command, the command connection removes the command name portion, loads the file containing the command, and passes the parameters to the command.
- Recognizes the ampersand (&) mark in a command line and displays a different prompt (**) when a continuation line is required.
- Displays error messages in the event of certain operator errors.

This is the user environment described in the iRMX 86 OPERATOR'S MANUAL. If it satisfies the needs of your application system, you can assign the Human Interface CLI to each operator as an initial program.

CUSTOMIZED INITIAL PROGRAMS

If the standard initial program does not meet your needs, you have the option of providing your own initial programs. These initial programs might be similar to the Human Interface CLI, or they might be completely different kinds of programs. 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. You specify this selection in the user description files maintained by the system manager (refer to the iRMX 86 CONFIGURATION GUIDE).

If you provide your own initial program, this program must obey the following rules:

- 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 86 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$SEND$CO$RESPONSE
  C$SEND$EO$RESPONSE
  C$SEND$COMMAND
  C$DELETE$COMMAND$CONNECTION

- If it does not create a command connection but still wishes to use the Human Interface system calls C$GET$PARAMETER and C$GET$CHAR, it must first invoke the C$SET$PARSE$BUFFER system call.

- If it receives an end-of-file indication from the terminal, it must terminate processing.

- It must invoke the Extended I/O System call EXIT$IO$JOB to terminate processing. It must not use the PL/M-86 or ASM86 RETURN statement for this purpose.

Refer to Chapter 8 for detailed descriptions of the Human Interface system calls mentioned in this section. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about the EXIT$IO$JOB system call.

***

CHAPTER 3
COMMAND PARSING

Whenever a Human Interface operator enters characters at a terminal to invoke a command, an initial program associated with that operator reads that 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.

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

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. 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.

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 iRMX 86 OPERATOR'S MANUAL 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 replaces the existing file with the new output. 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 iRMX 86 OPERATOR'S MANUAL 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 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 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 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.

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 entered in the command line.

PARSING THE COMMAND LINE

When a 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$GET$INPUT$PATHNAME
C$GET$OUTPUT$PATHNAME
C$GET$PARAMETER
C$GET$CHAR
```

Each of the first three system calls reads an entire parameter and causes 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 last system call, C$GET$CHAR, sees the parsing buffer as a string of characters. It reads a single character and causes the Human Interface to move the pointer to the next character. It does not understand the notion of quoting characters; therefore it does not remove the special meaning from quoted characters, nor does it skip over the quotes. Except for positioning the parsing pointer to a particular place in the buffer, C$GET$CHAR should not be used with the first three system calls.

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. Since the command line can contain multiple pathnames, you might have to invoke these system calls several times to obtain all the pathnames. However, the first call to C$GET$INPUT$PATHNAME reads the entire inpath-list (the list of pathnames separated by commas) into a buffer, moves the parsing pointer to the next parameter, and returns the first pathname to the command. Likewise, the first call to C$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 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$PATHNAME obtains both input pathnames (A and B) and returns the first one (A) to the caller. The first call to C$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.

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 86 <u>strings</u>. 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 command 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;

$include (hexcep.lit)
$include (iexioj.ext)
$include (hgticn.ext)
$include (hgtipn.ext)
$include (hgtocn.ext)
$include (hgtopn.ext)

DECLARE (input$pathname, output$pathname)  structure (
                                             length        byte,
                                             char (41)     byte ),
        output$prep     byte,
        (input$token, output$token)    word,
        excep      word,
        exitexcep  word;

/*  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 (exitexcep, 0, @excep);

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 (exitexcep, 0, @excep);
```

Figure 3-1. C$GET$INPUT$PATHNAME And C$GET$OUTPUT$PATHNAME Example

```
    /*  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 (exitexcep, 0, @excep);

    output$token = rq$C$get$output$connection (@output$pathname,
                                                output$prep, @excep);
    IF excep <> E$OK         THEN
        CALL rq$exit$io$job (exitexcep, 0, @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 (exitexcep, 0, @excep);

    END  /* DO WHILE */

/*  Finish I/O processing */
CALL rq$exit$io$job (exitexcep, 0, @excep);

END copy;
```

Figure 3-1. C$GET$INPUT$PATHNAME And C$GET$OUTPUT$PATHNAME Example (continued)

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 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 iRMX 86 OPERATOR'S MANUAL 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 <u>matches</u> 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 searchs the :PROG:PLM directory and returns the pathname of the next file that begins with the letter "A."

C$GET$OUTPUT$PATHNAME <u>generates</u> wild cards. Each time you call it, it compares the wild-carded output pathname with the wild-carded input pathname and with the most recent pathname returned by C$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 which 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
All	Cll
ADAM	CDAM
BETA	DETA
Bll	Dll

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 the 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 C). 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 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.                                                    *
***********************************************************************/

progl: DO;

$include (hexcep.lit)
$include (hgtpar.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$buf),
        excep WORD,
        (size$val, i) WORD;
```

Figure 3-2. C$GET$PARAMETER Example

```
/* Get the next parameter, if present */
IF (rq$C$get$parameter (@parameter$keyword, PARAMETER$KEYWORD$MAX,
                        @value$table, VALUE$TABLE$MAX,
                        0,0,
                        @excep) ) THEN
    IF (parameter$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
                   .
```

Figure 3-2. C$GET$PARAMETER Example (continued)

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 the invocation line to be 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.

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. Then, whenever it 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.

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.
- You must place a prefix character 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 again 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$GET$CHAR system call to parse the command line. This system call performs a single, simple operation. It returns a single character from the command line and moves the pointer to the next character. It does not understand the notion of parameters as explained earlier in this chapter. Nor does it understand wild-card characters or quoting characters.

C$GET$CHAR requires you to provide the parsing algorithm in your own program, because it makes no assumptions about the structure or order of parameters. However, by using C$GET$CHAR you can enforce any command syntax you choose.

Because C$GET$CHAR moves the pointer character by character, not parameter by parameter, you should take care when using C$GET$CHAR in the same program with C$GET$INPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and C$GET$PARAMETER. You must ensure that C$GET$CHAR leaves 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.

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

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 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 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).

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$NAME and the ampersand pathname separator (&) to access the directory in which it resides. For example, if "command-name" is the name received from C$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 AND MESSAGE PROCESSING

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.

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.

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 Chapter 7 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 only has to decide whether to abort or to continue with processing.

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", the system call obtains the connection to the existing file. If the operator enters "N" or "n", 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 Chapter 7 for more information.

EXAMPLE PROGRAM SCENARIO

A normal scenario for using C$GET$INPUT$CONNECTION and C$GET$OUTPUT$CONNECTION is as follows:

```
DO;

    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

UNTIL no more input and output pathnames remain
```

The program listing in Figure 3-1 shows an implementation of this scenario.

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 anything

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.

FORMATTING MESSAGES BASED ON EXCEPTION CODES

Whenever you include iRMX 86 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 86 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 iRMX 86 NUCLEUS 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 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$RESPONSE to write the information to the operator terminal. By using C$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.

***

CHAPTER 5 COMMAND PROCESSING

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 command line 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 iRMX 86 system calls appear in a program.

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.

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 86 Human Interface command (as described in the iRMX 86 OPERATOR'S MANUAL) or any command that you write.

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 a condition code informing 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 also 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 86 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.

DELETING THE COMMAND CONNECTION

After you have finished invoking commands programmatically, you must delete the command connection. The C$DELETE$COMMAND$CONNECTION system call performs this operation. You do not need to delete the command connection after each command invocation, because the command connection is re-usable. However, you should delete the command connection after performing all C$SEND$COMMAND operations. This frees the memory used by the data structures of the command connection.

EXAMPLE

Figure 5-1 contains an example of a program that uses C$CREATE$COMMAND$CONNECTION, SEND$COMMAND, and DELETE$COMMAND$CONNECTION. It invokes the Human Interface COPY command programmatically.

```
/***********************************************************************
*                                                                      *
*   This example demonstrates the use of the following Human Interface *
*   advanced standard 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 from within and then continue normal       *
*   processing.  The program is invoked with the command line:         *
*                                                                      *
*       PROG2                                                          *
***********************************************************************/

prog2:  DO;

$include (hexcep.lit)
$include (hcrccn.ext)
$include (hsndcmd.ext)
$include (hdlccn.ext)
$include (iexioj.ext)
$include (hgtincn.ext)
$include (hgtocn.ext)

DECLARE (ci$token, co$token, command$connection$token)  WORD,
        (excep, comexcep, exexcep)  WORD;
DECLARE output$prep  BYTE;
```

Figure 5-1. Command Connection Example

```
                    .
                    .
                    .

/* 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$job (excep, 0, exexcep);
co$token = rq$C$get$output$connection(@(4,':CO:'), output$prep, @excep);
IF excep <> E$OK        THEN
    CALL rq$exit$io$job (excep, 0, exexcep);

/* Create command connection */
command$connection$tok = rq$C$create$command$connection (@ci$token,
                                                          co$token, 0,
                                                          @excep);

/* Send command to copy files */
CALL rq$C$send$command (command$connection$tok,
                        @(23,'COPY :F1:OLD TO :F1:NEW'),
                        @comexcep, @excep);
IF excep <> E$OK        THEN
    CALL rq$exit$io$job (excep, 0, exexcep);

/* Delete command connection */
CALL rq$C$delete$command$connection (command$connection$tok, @excep);
IF excep <> E$OK        THEN
    CALL rq$exit$io$job (excep, 0, exexcep);

                    .
                    .         Rest of program
                    .

/* Finish I/O processing */
CALL rq$exit$io$job (excep, 0, @exexcep);

END prog2;
```

Figure 5-1. Command Connection Example (continued)

***

CHAPTER 6
PROGRAM CONTROL

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, the operator can enter a Control-C character. In the default case, the Control-C causes the Human Interface to abort the currently-executing command. However, you can override the default Control-C mechanism by providing your own code to process Control-C characters. This chapter discusses how to do this.

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.

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 an hour or more to create. Instead, the operator might want a Control-C to abort an individual editor command, but not abort the entire editor. 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 S$SPECIAL system call of the Extended I/O System to replace the Control-C semaphore. This system call is described in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL. However, it has three parameters that are important when changing the semaphore. They are:

connection	This parameter should contain the token for a connection to the operator's terminal.
function	This parameter should contain the value 6 to indicate the set signal character function.
data$ptr	This parameter should point to a structure of the following form:

```
DECLARE signal$pair  STRUCTURE(
     semaphore       WORD,
     character       BYTE);
```

where:

semaphore	A token for your new Control-C semaphore.
character	The character code for the Control-C character. If you use the ASCII code (03), the Operating System will place a unit in the semaphore when an operator enters Control-C. If you use the ASCII code plus 20H (23H), the Operating System clears out the terminal's input buffers in addition to placing the unit in the semaphore.

If your command task switches the Control-C semaphore, it must 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. Once the job for the initial command is deleted by the Human Interface, then Control-C once again becomes the default method for program control. The Human Interface reactivates Control-C by resetting a semaphore when the original command finishes. For example, once the text editor we used as an example terminates, then the Human Interface resets the semaphore so that Control-C becomes active.

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. Call S$ATTACH$FILE to obtain a connection to the terminal. Use logical name :CI: as the pathname parameter.

4. Call S$OPEN to open the connection to the terminal for reading only (mode 1).

5. If you plan to use a Control-C task, have the program call CREATE$TASK to start the Control-C task.

6. Call S$SPECIAL to switch the Control-C semaphore to the one just created. Use the token for the connection to the terminal as input.

7. Continue with command processing. If you are servicing the Control-C semaphore in-line, periodically check the semaphore (by calling RECEIVE$UNITS) to determine if it contains any units. If you obtain a unit 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 would be:

1. Call LOOKUP$OBJECT to obtain the token for the semaphore.

2. Do forever:

    a. Call RECEIVE$UNITS 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 that 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 guarantee a quick response to the Control-C in all cases.

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 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.

Therefore, unless you have a specific reason for using a Control-C task, this manual recommends that you use in-line code to service the Control-C semaphore.

***

CHAPTER 7 CREATING HUMAN INTERFACE COMMANDS

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 link your code.

To perform the operations described in this chapter you must have either an iAPX 86-based Microcomputer Development System (such as a Series III) or an iRMX 86-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 LINK86).

ELEMENTS OF A HUMAN INTERFACE COMMAND

This section discusses the rules that every command you write must obey. It also suggests some programming practices to make coding and using your command easier.

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 for this. 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$PATHNAME to obtain the first input pathname.
2. Call C$GET$OUTPUT$PATHNAME to obtain the preposition and first output pathname.
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 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.

AVOIDING THE USE OF CERTAIN SYSTEM CALLS

When you write the code for your Human Interface command, you can use any of the iRMX 86 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 86 reference manuals contain cautions concerning those system calls that you should avoid using.

In particular, avoid iRMX 86 objects (and their associated system calls) that, by their use, make your command immune to deletion. Regions and extension objects (described in the iRMX 86 NUCLEUS REFERENCE MANUAL) 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; also the operator's terminal may lock up when the command finishes processing.

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 iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL discusses I/O jobs.) Until the command finishes processing, the operator is unable to run any other commands. In order to finish processing 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 iRMX 86 EXTENDED I/O SYSTEM 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 omits the call to EXIT$IO$JOB, the operator might not be able to enter further commands.

INCLUDE FILES

When you write the source code for your commands, you can use $INCLUDE statements to include the following kinds of information: external declarations of system calls, literal definitions of exception codes, and common pieces of code that you declare.

As part of writing the code for your commands, you must declare each iRMX 86 system call as an external procedure. Instead of writing this code yourself, you can use the $INCLUDE statement to include this information from files on one of the iRMX 86 release diskettes. This diskette contains a file for each system call, with the external declaration of that system call as the contents of the file. To use these files, simply determine the system calls that your command uses and place into your source code $INCLUDE statements for the corresponding external declaration files.

You 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). The Include Files release diskette contains several files (one for each layer of the Operating System) consisting of LITERALLY statements. Each file defines all the iRMX 86 condition code mnemonics used in that layer. You should copy these files, delete entries if you can guarantee that the deleted exception codes will never appear, and use $INCLUDE statements to include them in the compilation of your command.

Refer to the iRMX 86 INSTALLATION GUIDE for information about the release diskettes and the files contained in them. Refer to the PL/M-86 USER'S GUIDE for information about the $INCLUDE statement.

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 86 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 LINK86, link the code to iRMX 86 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 LINK86 command is:

```
LINK86                            &
    command-pathname,             &
    :dir:HPIFC.LIB,               &
    :dir:LPIFC.LIB,               &
    :dir:EPIFC.LIB,               &
    :dir:IPIFC.LIB,               &
    :dir:RPIFC.LIB,               &
    :dir:other.lib                &
  TO   output-pathname            &
    PRINT(mapfile-pathname) SYMBOLCOLUMNS(2)            &
    OBJECTCONTROLS(PURGE)                               &
    BIND SEGSIZE(STACK(stacksize)) MEMPOOL(minsize,maxsize)
```

where:

command-pathname	Complete pathname of the file containing your compiled (or assembled) command. You can link in several files or libraries at this point, if necessary.
other.lib	Any other files or libraries that you need to link with your command.
output-pathname	Complete pathname of the file in which LINK86 places the linked command.
mapfile-pathname	Complete pathname of the file on which LINK86 places the link map.
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. Refer to the iRMX 86 PROGRAMMING TECHNIQUES manual for information about stack requirements of the system calls.
minsize maxsize	Minimum and maximum amount of dynamic memory, in bytes, required by the command. The command uses this memory when it creates iRMX 86 objects. The Human Interface uses the minsize and maxsize values when it creates 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.

This command produces relocatable code that the Operating System can load into any available memory. If you require your command to be available as absolute code, you can use LINK86 and LOC86 to produce this code. Refer to the iAPX 86, 88 FAMILY UTILITIES USER'S GUIDE for more information about LINK86 and LOC86. If you require absolute code for your commands, you must also configure the Operating System in such a way that it reserves the memory locations required by the command. If it does not, the command, when loaded into the system, could overwrite Operating System or user information. Refer to the iRMX 86 CONFIGURATION GUIDE for more information about Operating System configuration.

If you are using an iRMX 86-based system to compile and link your command, the command is now ready for execution. An operator can invoke the command by entering the pathname of the file containing the linked command (the output-pathname in the LINK86 command).

If you are using a Microcomputer Development System to compile or link your command, you must connect the development system to your iRMX 86 application system via the monitor and use the Human Interface UPCOPY command to copy the linked command from the development system disk to an iRMX 86 secondary storage device. The UPCOPY command is described in the iRMX 86 OPERATOR'S MANUAL. After you transfer the linked command to an iRMX 86 secondary storage device, an operator can invoke the command by entering its pathname.

***

CHAPTER 8 HUMAN INTERFACE SYSTEM CALLS

The Human Interface system calls described in this chapter are presented in alphabetical sequence without regard to functional organization. A functional grouping of the calls according to type is provided in the System Call Dictionary in Table 8-1. For each call, the information is organized into the following categories:

- Brief functional description.
- Calling sequence format.
- Input parameter definitions, if applicable.
- Output parameter definitions, if applicable.
- Considerations and consequences of call usage.
- Potential exception codes, and their possible causes.

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

If you are a new user of the Human Interface calls, it is suggested that you review the parsing considerations in Chapter 3 before writing your source code. You should also review the format of the released Human Interface commands. They are described in the iRMX 86 OPERATOR'S MANUAL.

This chapter assumes that you are familiar with several terms and concepts that are common to the iRMX 86 Operating System. If you are not, you should read INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM and the chapters in the iRMX 86 NUCLEUS REFERENCE MANUAL that refer to the terms "memory pool" and "catalog."

Table 8-1. System Call Dictionary

System Call	Synopsis	Page
	I/O Processing Calls	
C$GET$INPUT$CONNECTION	Return an EIOS connection for the specified input file.	8-15
C$GET$OUTPUT$CONNECTION	Return an EIOS connection for the specified output file.	8-25
	Command Parsing Calls	
C$GET$CHAR	Get a character from the command line	8-11
C$GET$INPUT$PATHNAME	Parse the command line and return an input pathname.	8-20
C$GET$PARAMETER	Parse the command line for the next parameter and return it as a keyword name and a value.	8-34
C$GET$OUTPUT$PATHNAME	Parse the command line and return an output pathname.	8-31
C$SET$PARSE$BUFFER	Parse a buffer other than the current command line.	8-51
C$GET$COMMAND$NAME	Return the command name by which the the current command was invoked	8-13
	Message Processing Calls	
C$FORMAT$EXCEPTION	Create a default message for an exception code and place it in a user buffer.	8-9
C$SEND$CO$RESPONSE	Send a message to the command output (CO) and read a response from the command input (CI).	8-45
C$SEND$EO$RESPONSE	Send a message to the operator's terminal and return a response from that terminal.	8-48

Table 8-1. System Call Dictionary (continued)

System Call	Synopsis	Page
Command Processing Calls		
C$CREATE$COMMAND$CONNECTION	Create a command connection and return a token.	8-4
C$DELETE$COMMAND$CONNECTION	Delete a specific command connection.	8-8
C$SEND$COMMAND	Concatenate command lines into the data structure created by CREATE$COMMAND$CONNECTION and then invoke the command.	8-38

C$CREATE$COMMAND$CONNECTION

C$CREATE$COMMAND$CONNECTION, a command processing call, creates an iRMX 86 object called a command connection that is required in order to invoke commands programmatically.

```
command$conn = RQ$C$CREATE$COMMAND$CONNECTION(default$ci, default$co,
                                              flags, except$ptr);
```

INPUT PARAMETERS

default$ci	A TOKEN for a connection that is used as the :CI: (console input) for any commands you invoke using this command connection.
default$co	A TOKEN for a connection that is used as the :CO: for any commands you invoke using this command connection.
flags	A WORD used to indicate that the Human Interface should return an E$ERROROUTPUT exception code if the system call C$SEND$EO$RESPONSE is used by any task. If the user wants the exception code, then the parameter is set to one (1); otherwise, the parameter must equal zero (0).

OUTPUT PARAMETERS

command$conn	A TOKEN which receives a token for the new command connection.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

You can use this call when you want to invoke a command programmatically instead of interactively. It provides a place to store command lines until the command invocation is complete.

The call creates an iRMX 86 object called a command connection and returns a token for that command connection. The C$SEND$COMMAND system call can use this token to send command lines to the command connection, where they are stored until the command invocation is complete. The command connection also defines default :CI: and :CO: connections that are used by any commands invoked via this command connection.

Although a job can contain multiple command connections, the tasks in a job cannot create command connections simultaneously. Attempts to do this result in an E$CONTEXT exception code. Therefore, it is advisable for one task to create the command connections for all tasks in the job.

A possible application where the parameter "flags" might be set to one is when you want to write a custom CLI to perform batch jobs in the background. When any of the background batch jobs attempt to communicate with the terminal through C$SEND$EO$RESPONSE, the Human Interface issues an exception code. In this way, the Human Interface keeps all the jobs in the background. Note--the Human Interface CLI does not provide resident background or batch processing capability.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$ATTACHED	While creating a STREAM file, the Extended I/O System was unable to attach the :STREAM: device because another task had already invoked a Basic I/O system call to attach the :STREAM: device.
E$CONTEXT	At least one of the following is true: • Two command connections were being created simultaneously by two tasks in the same job. • The calling task's job is not an I/O job.(Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.)
E$DEV$DETACHING	The :STREAM: device, the default$ci device, or the default$co device was in the process of being detached.
E$DEVFD	The Extended I/O System attempted the physical attachment of the :STREAM: device. This device had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible. The Operating System would not have returned this exception code if the :STREAM: device had been properly configured.
E$EXIST	The default$ci or default$co parameter is not a token for an existing job.
E$FNEXIST	The :STREAM: file does not exist or is marked for deletion.

E$IFDR	The Extended I/O System attempted to obtain information about the default$ci or default$co connection. However, the request for information resulted in an invalid file driver request.
E$INVALID$FNODE	The fnode associated with the specified file is invalid. Delete the file.
E$IOMEM	The Basic I/O System job does not currently have a block of memory large enough to allow the Human Interface to create a stream file.
E$LIMIT	At least one of the following is true: ● The object directory of the calling task's job has already reached the maximum object directory size. ● The calling task's job has exceeded its object limit. ● The calling task's job (or that job's default user object) is already involved in 255 (decimal) I/O operations. ● The calling task's job is not I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.)
E$LOG$NAME$NEXIST	The call was unable to find the logical name :STREAM: in the object directories of the local job, the global job, or the root job.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$NO$PREFIX	The calling task's job does not have a valid default prefix.
E$NOT$CONNECTION	The default$ci or default$co parameter is a token for an object, not a connection to a file.
E$NOT$LOG$NAME	The logical name :STREAM: refers to an object that is not a file or device connection.
E$NO$USER	The calling task's job does not have a valid default user object.

E$PARAM	The system call forced the Extended I/O System to attempt the physical attachment of the :STREAM: device, which had formerly been only logically attached. In the process, the Extended I/O System found that the stream file driver is not properly configured into your system, so the physical attachment is not possible.
E$SUPPORT	The default$ci or default$co connection was not created by this job.

C$DELETE$COMMAND$CONNECTION

C$DELETE$COMMAND$CONNECTION, a command processing call, deletes a command connection object and frees the memory used by the command connection's data structures.

```
CALL RQ$C$DELETE$COMMAND$CONNECTION(command$conn, except$ptr);
```

INPUT PARAMETER

command$conn	A TOKEN for a valid command connection.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

This call deletes a command connection object previously defined in a C$CREATE$COMMAND$CONNECTION call and releases the memory used by the command connection's data structures.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$EXIST	The command$conn parameter is not a token for an existing object.
E$TYPE	The command$conn parameter is a token for an object that is not a command connection object.

C$FORMAT$EXCEPTION

C$FORMAT$EXCEPTION, a message processing call, creates a default message for a given exception code and writes that message into a user-provided string.

```
CALL RQ$C$FORMAT$EXCEPTION(buff$p, buff$max, exception$code,
                           reserved$byte, except$ptr);
```

INPUT PARAMETERS

buff$max	A WORD that specifies the maximum number of bytes that may be contained in the string pointed to by buff$p.
exception$code	A WORD containing the exception code value for which a message is to be created.
reserved$byte	A BYTE reserved for future use. Its value must be one (1).

OUTPUT PARAMETER

buff$p	A POINTER to a STRING into which the Human Interface concatenates the formatted exception message.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

C$FORMAT$EXCEPTION causes the Human Interface to create a message for the exception code. The message consists of the exception code value and exception code mnemonic in the following format:

```
value : mnemonic
```

where the mnemonics are provided by the Human Interface from an internal table and are listed in Appendix B of this manual.

The call concatenates the message to the end of the string pointed to by the buff$p pointer and updates the count byte to reflect the addition. If a string is not already present in the buffer, the first byte of the buffer must be a zero. The message added by C$FORMAT$EXCEPTION will not be longer than 30 characters (not including the length byte).

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$PARAM	An undefined exception code value was specified.
E$STRING	The message to be returned exceeds the length limit of 255 characters.
E$STRING$BUFFER	The buffer pointed to by the buff$p parameter is not large enough to contain the exception message.

C$GET$CHAR

C$GET$CHAR, a command parsing call, gets a character from the parsing buffer.

```
    char = RQ$C$GET$CHAR(except$ptr);
```

OUTPUT PARAMETERS

char	A BYTE in which the Human Interface places the next character of the parsing buffer. A null (00H) character is returned when parsing buffer's pointer is at the end of the buffer.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

When an operator invokes a command, the command's parameters are placed in a parsing buffer. The C$GET$CHAR system call gets a single character from that buffer and moves the parsing pointer to the next character. Consecutive calls to C$GET$CHAR return consecutive characters from the parsing buffer.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$CONTEXT	The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E$LIMIT	At least one of the following situations occurred. • The object directory of the calling task's job has already reached the maximum object directory size. • The calling task's job has exceeded its object limit.

	• The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.

C$GET$COMMAND$NAME

C$GET$COMMAND$NAME, a command parsing call, obtains the pathname of the command that the operator used when invoking the command.

```
CALL RQ$C$GET$COMMAND$NAME (path$name$p, name$max, except$ptr);
```

INPUT PARAMETER

name$max	A WORD that specifies the length in bytes of the string pointed to by the path$name$p parameter.

OUTPUT PARAMETERS

path$name$p	A POINTER to a STRING that receives the name of the command (the last component of the pathname).
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

If a command needs to know the name under which it was invoked, the C$GET$COMMAND$NAME returns this information. This information is available to each command and is stored in a buffer that is separate from the parsing buffer. Therefore, calling C$GET$COMMAND$NAME does not obtain information from the parsing buffer, nor does it move the parsing pointer.

If the operator invokes the command without specifying a logical name, the Human Interface automatically searches a number of directories for the command. In such cases, the value returned by C$GET$COMMAND$NAME also includes the directory name (such as :SYSTEM:, :PROG:, or :$:) as a prefix to the command name.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$LIMIT	The calling task's job was not created by the Human Interface.

E$PATHNAME$SYNTAX	The specified pathname contains invalid characters.
E$STRING$BUFFER	The buffer pointed to by the path$name$p parameter is not large enough to contain the command name.
E$TIME	The calling task's job was not created by the Human Interface.

C$GET$INPUT$CONNECTION

C$GET$INPUT$CONNECTION, an I/O processing call, returns an Extended I/O System connection to the specified input file.

```
connection = RQ$C$GET$INPUT$CONNECTION(path$name$p, except$ptr);
```

INPUT PARAMETER

path$name$p	A POINTER to a STRING containing the pathname of the file to be accessed.

OUTPUT PARAMETERS

connection	A TOKEN in which the Operating System returns the token for the connection to the specified pathname.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

C$GET$INPUT$CONNECTION obtains a connection to the specified file. This connection is open for reading and has the following attributes:

- Read only
- Accessible to all users
- Has two 1024-byte buffers (The buffer size may be different than the default value of 1024 bytes.)

C$GET$INPUT$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever the Operating System encounters an exceptional condition. The exceptional condition that triggers the error message can either be one of those listed for C$GET$INPUT$CONNECTION or it can be one of those associated with the Extended I/O System calls S$ATTACH$FILE and S$OPEN. The following messages can occur:

- <pathname>, file does not exist

  The input file does not exist.

- <pathname>, invalid file type

  The input file was a data file and a directory was required, or vice versa.

- <pathname>, invalid logical name

  The input pathname contains a logical name that is longer than 12 characters, that contains unmatched colons, or that contains invalid characters.

- <pathname>, logical name does not exist

  The input pathname contains a logical name that does not exist.

- <pathname>, READ access required

  The user does not have read access to the input file.

- <pathname>, <exception value>:<exception mnemonic>

  An exceptional condition occurred when C$GET$INPUT$CONNECTION attempted to obtain the input connection. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered. Refer to "Exception Codes" in this call description and to the descriptions of S$ATTACH$FILE and S$OPEN in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The device containing the file specified in the path$name$p parameter is already attached.
E$CONTEXT	At least one of the following is true: • The calling task's job is not an I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.) • The calling task's job was not created by the Human Interface.
E$DEV$DETACHING	The device specified in the path$name$p parameter is in the process of being detached.

E$DEVFD	The call attempted the physical attachment of a device that had formerly been only logically attached. In the process, the call found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	The specified device does not exist.
E$FACCESS	The specified connection does not have read access to the file.
E$FNEXIST	At least one of the following is true: ● The target file does not exist or is marked for deletion. ● While attaching the file pointed to by the path$name$p parameter, the call attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
E$IO$HARD	While attempting to access the file specified in the path$name$p parameter, the call detected a hard I/O error. This means that another call is probably useless.
E$IOMEM	While attempting to create a connection, the call needed memory from the Basic I/O subsystem's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
E$IO$OPRINT	While attempting to access the file specified in the path$name$p parameter, the call found that the device was off-line. Operator intervention is required when given this code.

E$IO$SOFT	While attempting to access the file specified in the path$name$p parameter, the call detected a soft I/O error. It tried the operation again but was unsuccessful. Another try might be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred while this call tried to access the file given in the path$name$p parameter.
E$LIMIT	At least one of the following is true: • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$LOG$NAME$-NEXIST	The pathname for the specified device contains an explicit logical name. The call was unable to find this name in the object directories of the local job, the global job, or the root job.
E$LOG$NAME$-SYNTAX	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	The specified device was off-line.
E$MEM	The memory available to the calling task's job is not sufficent to complete the call.
E$NO$PREFIX	The calling task's job does not have a valid default prefix.
E$NOT$LOG$NAME	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
E$NO$USER	The calling task's job does not have a valid default user.
E$PARAM	At least one of the following is true: • The system call forced the Extended I/O System to attempt the physical attachment of the device referenced by the path$name$p parameter. This device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible.

	• The connection to the specified file cannot be opened for reading.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
E$SHARE	The files sharing attribute currently does not allow new connections to the file to be opened for reading.
E$STREAM$SPECIAL	The call attempted to attach a stream file and in so doing issued an invalid stream file request.

C$GET$INPUT$PATHNAME

C$GET$INPUT$PATHNAME, a command parsing call, gets a pathname from the list of input pathnames in the parsing buffer.

```
   CALL RQ$C$GET$INPUT$PATHNAME(path$name$p, path$name$max, except$ptr);
```

INPUT PARAMETER

path$name$max	A WORD that specifies the length in bytes of the string pointed to by the path$name$p parameter. The maximum length that you can specify is 256 bytes (255 characters for the pathname and one byte for the count).

OUTPUT PARAMETERS

path$name$p	A POINTER to a STRING which receives the next pathname in the pathname list. A zero-length string indicates that there are no more pathnames.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

The first call to C$GET$INPUT$PATHNAME retrieves the entire input pathname list and moves the parsing pointer to the next parameter. C$GET$INPUT$PATHNAME stores the list in an internal buffer and returns the first pathname to the string pointed to by the path$name$p parameter. Succeeding calls to C$GET$INPUT$PATHNAME return additional pathnames from the input pathname list but do not move the parsing pointer. C$GET$INPUT$PATHNAME denotes the end of the pathname list by returning a zero-length string.

C$GET$INPUT$PATHNAME accepts wild-card characters in the last component of a pathname. It treats a wild-carded pathname as a list of pathnames. To obtain each pathname, it searches in the parent directory of the wild-carded component, comparing the wild-carded name with the names of all files in the directory. It returns the next pathname that matches.

The pathname returned by C$GET$INPUT$PATHNAME can be used for any purpose. However, it is most often used in a call to C$GET$INPUT$CONNECTION, to obtain a connection.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The device containing the file pointed to by the path$name$p parameter is already attached.
E$CONTEXT	At least one of the following is true: • The calling task's job is not an I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about I/O jobs.) • The task called C$GET$OUTPUT$PATHNAME before calling C$GET$INPUT$PATHNAME.
E$DEV$DETACHING	The device pointed to by the path$name$p parameter is in the process of being detached.
E$DEVFD	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	At least one of the following is true: • The connection to the parent directory of the file pointed to by the path$name$p parameter, is not a token for the existing job. • The calling task's job was not created by the Human Interface.
E$FACCESS	The connection used to open the directory does not have read access to the directory.
E$FLUSHING	The device containing the directory was in the process of being detached.

E$FNEXIST	At least one of the following is true: • The target file does not exist or is marked for deletion. • While attaching the parent directory of the file pointed to by the path$name$p parameter, the I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$IFDR	The specified file is a stream or physical file.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
E$IO$HARD	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, the call detected a hard I/O error. This means that another call is probably useless.
E$IOMEM	While attempting to create a connection, this call needed memory from the Basic I/O subsystem's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
E$IO$OPRINT	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, this call detected that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value E$IO$NOT$READY for this code.

E$IO$SOFT	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, this call detected a soft I/O error. It tried the operation again, but was unsuccessful. Another try might be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred while this call tried to access the parent directory of the file pointed to by the path$name$p parameter.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$LIST	The last value of the input pathname list is missing. For example, "ABLE,BAKER," has no value following the second comma.
E$LOG$NAME$-NEXIST	The pathname for the specified device contains an explicit logical name. The call was unable to find this name in the object directory of the local job, the global job, or the root job.
E$LOG$NAME$-SYNTAX	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	The specified device was off-line.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$NO$PREFIX	The calling task's job does not have a valid default prefix.
E$NOT$LOG$NAME	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
E$NO$USER	The calling task's job does not have a valid default user object.

E$PARAM	At least one of the following is true: • The Extended I/O System attempted the physical attachment of the device pointed to by the path$name$p parameter. This device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible. • The connection to the parent directory cannot be opened for reading.
E$PARSE$TABLES	The call detected an error in an internal table used by the Human Interface.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
E$SHARE	The connection to the parent directory cannot be opened for reading.
E$STREAM$-SPECIAL	The Extended I/O System attempted to attach a stream file and in so doing issued an invalid stream file request.
E$STRING	The pathname to be returned exceeds the length limit of 255 characters.
E$STRING$BUFFER	The buffer pointed to by the path$name$p parameter was not large enough for the pathname to be returned.
E$SUPPORT	This call attempted to read the parent directory of the pathname pointed to by the path$name$p parameter. However, the file driver corresponding to that directory does not support this operation.
E$WILD$CARD	The pathname to be returned contains an invalid wild-card specification.

C$GET$OUTPUT$CONNECTION

C$GET$OUTPUT$CONNECTION, an I/O processing call, parses the command line and returns an Extended I/O System connection referring to the requested output file.

```
connection = RQ$C$GET$OUTPUT$CONNECTION(path$name$p, preposition,
                                         except$ptr);
```

INPUT PARAMETERS

path$name$p	A POINTER to a STRING containing the pathname of the file to be accessed.
preposition	A BYTE that defines which preposition to use to create the output file. Use one of the following values to specify the preposition mode:

Value	Meaning
0	Use same preposition as was returned by the last C$GET$OUTPUT$PATHNAME call
1	TO
2	OVER
3	AFTER
4-255	Undefined, results in an error

OUTPUT PARAMETERS

connection	A TOKEN in which the Human Interface returns a token for the connection to the output file.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

C$GET$OUTPUT$CONNECTION obtains a connection to the specified file. This connection is open for writing and has the following attributes:

- Write only
- Accessible to all

If the call to C$GET$OUTPUT$CONNECTION specifies the TO preposition and the output file already exists, C$GET$OUTPUT$CONNECTION issues the following message to the terminal (:CO:):

    <pathname>, already exists, OVERWRITE?

If the operator enters Y, y, R, or r, C$GET$OUTPUT$CONNECTION returns a connection to the existing file, allowing the command to write over the file. Any other response causes C$GET$OUTPUT$CONNECTION to generate an E$FILEACCESS exception code.

C$GET$OUTPUT$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever an exceptional condition occurs. The exceptional condition that triggers the error message can be either one of those listed for C$GET$OUTPUT$CONNECTION or one of those associated with an Extended I/O System call. The following messages can occur:

- <pathname>, DELETE access required

  The user does not have delete access to an existing file.

- <pathname>, directory ADD entry access required

  The user does not have add entry access to the parent directory.

- <pathname>, file does not exist

  The output file does not exist.

- <pathname>, invalid file type

  The output file was a data file and a directory was required, or vice versa.

- <pathname>, invalid logical name

  The output pathname contains a logical name that is longer than 12 characters, that contains unmatched colons, or that contains invalid characters.

- <pathname>, logical name does not exist

  The output pathname contains a logical name that does not exist.

- <pathname>, <exception value>:<exception mnemonic>

  An exceptional condition occurred when C$GET$OUTPUT$CONNECTION attempted to obtain the input connection. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered. Refer to "Exception Codes" in this call description and to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The Extended I/O System was unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	The calling task's job was not created by the Human Interface.
E$DEV$DETACHING	The device referred to by the path$name$p parameter was in the process of being detached.
E$DEVFD	The call attempted the physical attachment of a device that had formerly been only logically attached. In the process, the call found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	The connection parameter for the device containing that file is not a token for an existing object.
E$FACCESS	At least one of the following is true: ● The default user for the calling task's job did not have update access to an existing file and/or add-entry access to the parent directory. ● The TO or OVER preposition was specified and the default user for the calling task's job did not have the ability to truncate the file.

E$FNEXIST	At least one of the following is true: • The target file does not exist or is marked for deletion. • While attaching the file pointed to by the path$name$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$IFDR	The call requested information about the specified file, but the request was an invalid file driver request.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
E$IO$HARD	While attempting to access the file specified in the path$name$p parameter, the call detected a hard I/O error. This means that another try is probably useless.
E$IOMEM	While attempting to create a connection, this call needed memory from the Basic I/O subsystem's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
E$IO$OPRINT	While attempting to access the file specified in the path$name$p parameter, the call detected that the device was off-line. Operator intervention is required.

E$IO$SOFT	While attempting to access the file specified in the path$name$p parameter, the call detected a soft I/O error. It tried the operation again but was unsuccessful. Another try might be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred while this call tried to access the file given in the path$name$p parameter.
E$IO$WRPROT	While attempting to obtain an input connection to the file specified in the path$name$p parameter, this call found that the volume containing the file is write-protected.
E$LIMIT	At least one of the following is true: • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about I/O jobs.)
E$LOG$NAME$-NEXIST	The specified pathname contains an explicit logical name. The call was unable to find this name in the object directory of the local job, the global job, or the root job.
E$LOG$NAME$-SYNTAX	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains unmatched colons, is longer than 12 characters, or contains invalid characters.
E$MEDIA	The specified device was off-line.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$NO$PREFIX	The calling task's job does not have a valid default prefix.
E$NOT$LOG$NAME	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
E$NO$USER	The calling task's job does not have a valid default user object.

E$PARAM	The system call forced the Extended I/O System to attempt the physical attachment of the device referenced by the path$name$p parameter. The device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
E$PREPOSITION	One of the following is true: • The command line contained an invalid preposition value (a value greater than 3). • The command line contained a zero as the preposition value. This indicated that the same preposition was to be used as in the last C$GET$OUTPUT$PATHNAME call. However, this is the first call to C$GET$OUTPUT$PATHNAME.
E$SHARE	The new connection cannot be opened for writing.
E$SPACE	One of the following is true: • The volume is full. • The volume already contains the maximum number of files.
E$STREAM$ SPECIAL	The Extended I/O System attempted to attach a stream file and in so doing issued an invalid stream file request.

C$GET$OUTPUT$PATHNAME

C$GET$OUTPUT$PATHNAME, a command parsing call, gets a pathname from the list of output pathnames in the parsing buffer.

```
    preposition = RQ$C$GET$OUTPUT$PATHNAME(path$name$p, path$name$max,
                                           default$output$p, except$ptr);
```

INPUT PARAMETERS

path$name$max	A WORD that specifies the length in bytes of the string pointed to by the path$name$p parameter. The maximum length that you can specify is 256 bytes (255 characters for the pathname and one byte for the count).
default$output$p	A POINTER to a STRING containing the command's default standard output. If the first invocation of this system call does not encounter a TO/OVER/AFTER preposition, the text of this parameter will be used as though it had appeared in the command line. The text must specify TO, OVER, or AFTER for the output mode. Examples: TO :CO: or TO :LP:.

OUTPUT PARAMETERS

preposition — A BYTE describing the preposition type that C$GET$OUTPUT$PATHNAME encountered. You can pass this value to C$GET$OUTPUT$CONNECTION when obtaining an output connection to the file. The value will be one of the following:

Value	Meaning
1	TO
2	OVER
3	AFTER

path$name$p	A POINTER to a STRING that receives the next pathname in the pathname list.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

You should not call C$GET$OUTPUT$PATHNAME before first calling C$GET$INPUT$PATHNAME.

The first call to C$GET$OUTPUT$PATHNAME retrieves the preposition (TO/OVER/AFTER) and the entire output pathname list; it then moves the parsing pointer to the next parameter. If the parsing buffer does not contain a preposition and pathname list, C$GET$OUTPUT$PATHNAME uses the default pointed to by the default$output$p parameter (and does not move the parsing pointer). After retrieving the pathname list, C$GET$OUTPUT$PATHNAME stores it in an internal buffer, returns the first pathname in the string pointed to by the path$name$p parameter, and returns the preposition in the preposition parameter. Succeeding calls to C$GET$OUTPUT$PATHNAME return additional pathnames from the output pathname list (as well as the preposition), but they do not move the parsing pointer. C$GET$INPUT$PATHNAME denotes the end of the pathname list by returning a zero-length string in the string pointed to by path$name$p.

C$GET$OUTPUT$PATHNAME accepts wild-card characters in the last component of a pathname. It generates each output pathname based on this wild-carded pathname, the corresponding wild-carded pathname that was input to C$GET$INPUT$PATHNAME, and the most recent input pathname returned by C$GET$INPUT$PATHNAME.

The pathname returned by C$GET$OUTPUT$PATHNAME can be used for any purpose. However, it is most often used in a call to C$GET$OUTPUT$CONNECTION to obtain a connection to the file. In such a case, C$GET$OUTPUT$CONNECTION processes the TO/OVER/AFTER preposition. If the pathname is used as input to a system call other than C$GET$OUTPUT$CONNECTION, the interpretation of the TO/OVER/AFTER preposition is the user's responsibility.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$CONTEXT	The calling task's job was not created by the Human Interface.
E$DEFAULT$SO	The default output string pointed to by default$output$p contained an invalid preposition or pathname.

E$LIMIT	At least one of the following is true: • The calling task's job has already reached its limit. • The calling task's job was not created by the Human Interface.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
E$STRING	The pathname to be returned exceeds the length limit of 255 characters.
E$STRING$-BUFFER	The buffer pointed to by the path$name$p parameter was not large enough for the pathname to be returned.
E$UNMATCHED$-LISTS	The numbers of files in the input and output lists are not same.
E$WILDCARD	The output pathname contains an invalid wild-card specification.

C$GET$PARAMETER

GET$PARAMETER, a command parsing call, gets a parameter from the parsing buffer.

```
more = RQ$C$GET$PARAMETER(name$p, name$max, value$p, value$max,
                            index$p, predict$list$p, except$ptr);
```

INPUT PARAMETERS

name$max	A WORD that specifies the length in bytes of the string pointed to by the name$p parameter. The maximum length is 256 bytes (255 characters for the name and one byte for the count).
value$max	A WORD that specifies the length in bytes of the string pointed to by the value$p parameter. The maximum length is 65535 decimal bytes.
predict$list$p	A POINTER to a STRING$TABLE, as described in Appendix C, that specifies the values that this system call accepts as prepositions. The predict$list$p POINTER should be zero if you do not intend to retrieve parameters that use prepositions.

OUTPUT PARAMETERS

more	A BYTE value that indicates whether or not the current call to C$GET$PARAMETER returned a parameter. A value of 00h indicates that there are no more parameters (and that no parameter was returned); a value of OFFh indicates that a parameter was returned.
name$p	A POINTER to a STRING that receives the keyword portion of the parameter. If this parameter does not contain a keyword portion, the Human Interface returns a null (zero-length) string.
value$p	A POINTER to a STRING$TABLE, as described in Appendix C, that receives the value portion of the parameter. If the value portion contains a list of values separated by commas, the Human Interface returns the values to the string table one value per string.

index$p	A POINTER to a BYTE that receives the index to the list of prepositions pointed to by predict$list$p. This index identifies the name$p keyword as a preposition and identifies it out of the list of possible prepositions. If the predict$list$p list is empty, or if the keyword name is not contained in the predict$list$p list, the system call returns a value of zero for the index. That is, the index will be non-zero only if a keyword exists and it is one of the prepositions in the predict$list$p list.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

C$GET$PARAMETER retrieves one parameter from the parsing buffer and moves the parsing pointer to the next parameter. The parameter can be one of the following:

- keyword/value-list parameter using parentheses
- keyword/value-list parameter using an equal sign
- keyword/value-list parameter with the keyword as a preposition
- value-list without a keyword

A description of the types, format, and syntax of acceptable parameters is provided in Chapter 3.

C$GET$PARAMETER places the keyword portion of the parameter in the string pointed to by name$p; it places the keyword list in the string table pointed to by value$p.

Without input from you, C$GET$PARAMETER cannot determine whether groups of characters separated by spaces are separate parameters or a single parameter that uses a preposition. C$GET$PARAMETER uses the list of prepositions that you supply in the string table pointed to by predict$list$p to determine the prepositions that can appear. When C$GET$PARAMETER retrieves a parameter, it obtains from the parsing buffer the next group of characters that are separated by spaces. Then it checks those characters against those in the predict$list$p list. If the characters match one of the values in the list, C$GET$PARAMETER realizes that the characters represent a preposition and not an entire parameter; it then obtains the next group of characters separated by spaces as the value portion of the parameter.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.

E$CONTEXT	The calling task's job was not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E$CONTINUED	The call found a continuation character in the parse buffer. Command lines should not contain continuation characters.
E$LIMIT	At least one of the following is true: ● The calling task's job has already reached its object limit. ● The calling task's job was not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E$LIST	At least one of the following is true: ● The parameter contains an unmatched parenthesis. ● A value in the value list is missing or an improper value was entered. Examples of both these conditions follow:

Value	Comments
A,B,	No value following second comma.
A,B=C,D	The equal sign can not be used unless it is between quotes: 'B=C' is valid.
A,B(C,E),F	The parentheses can not be used in a value unless it is between quotes or set off by commas. A,B,(C,E),F is valid.

E$LITERAL	The call found a literal (quoted string) in the parsing buffer with no closing quote. This condition should not occur in the command line buffer.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$PARAM	The predict$list$p parameter pointed to a string table, but the index$p parameter was set to zero (0).

E$PARSE$TABLES	The call found an error in an internal table used by the Human Interface.
E$SEPARATOR	The call found an invalid command separator in the parsing buffer. This condition should not occur in the command line buffer. The following is a list of invalid command separators: ><, <>, ||, |, [, and].
E$STRING	The string to be returned as the parameter name or one of the parameter values exceeds the length limit of 255 characters.
E$STRING$BUFFER	The string to be returned as the parameter name or one of the parameter values exceeds the buffer size provided in the call.

C$SEND$COMMAND

C$SEND$COMMAND, a command processing call, sends command lines to a command connection created by C$CREATE$COMMAND$CONNECTION and, when the command is complete, invokes the command.

```
CALL RQ$C$SEND$COMMAND(command$conn, line$p, command$except$ptr,
                           except$ptr);
```

INPUT PARAMETERS

command$conn	A TOKEN for the command connection that receives the command line.
line$p	A POINTER to a STRING containing a command line to execute.

OUTPUT PARAMETERS

command$except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code indicating the status of the invoked command. This parameter is undefined if an exceptional condition code is returned in the word pointed to by except$ptr.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code indicating the status of the C$SEND$COMMAND system call.

DESCRIPTION

You can use this system call when you want to invoke a command programmatically instead of interactively. It stores a command line in the command connection created by the C$CREATE$COMMAND$CONNECTION call, concatenates the command line with any others already stored there, and (if the command invocation is complete) invokes the command. The command can be any standard Human Interface command (as described in the iRMX 86 OPERATOR'S MANUAL) or a command that you create.

As described in greater detail in Chapter 3, a command invocation can contain several continuation marks. The continuation mark (&) indicates that the command line is continued on the next line. If the command line sent by C$SEND$COMMAND is continued on another line (that is, contains a continuation mark), the Human Interface returns an E$CONTINUED exception code and does not invoke the command. You can then call C$SEND$COMMAND any number of times to send the continuation lines.

C$SEND$COMMAND concatenates the original command line and all continuation lines into a single command line in the command connection. It removes all continuation marks and all comments from this ultimate command line.

When the command invocation is complete (that is, the line sent by C$SEND$COMMAND does not contain a continuation mark) the Human Interface parses the command pathname from the command line. If no exception conditions halt the process at this point, the Human Interface requests the Application Loader to load and execute the command.

An Application Loader call creates an I/O job for the command. Then the Application Loader validates the header, group definition and segment definition records of the command's object file. Refer to the 8086 FAMILY UTILITIES USER'S GUIDE for explanations of segments, groups and object file formats.

C$SEND$COMMAND returns two condition codes: one for the C$SEND$COMMAND call and one for the invoked command. The word pointed to by the except$ptr parameter returns the C$SEND$COMMAND conditions, as described under the EXCEPTION CODES heading in this command description. The word pointed to by the command$except$ptr returns the invoked command's condition codes; the values returned depend on the command invoked. The E$CONTROL$C exception code can be returned at either place.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The Extended I/O System was unable to attach the device containing the object file because the Basic I/O System has already attached the device.
E$BAD$GROUP	The object file represented by the command's pathname contained an invalid group definition record.
E$BAD$HEADER	The object file represented by the command's pathname does not begin with a header record for a loadable object module.
E$BAD$SEGMENT	The object file represented by the command's pathname contains an invalid segment definition record.
E$CHECKSUM	At least one record of the object file represented by the command's pathname contains a checksum error. This situation could occur if the CHECKSUM amount calculated during the read operation did not match the CHECKSUM field of the record being read.
E$CONTEXT	The calling task's job was not created by the Human Interface.

E$CONTINUED	The Operating System detected a continuation character while scanning the command line pointed to by the line$p parameter. This condition should occur if the command line is to continue on the next line.
E$DEV$DETACHING	The device containing the object file was in the process of being detached.
E$DEVFD	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$EOF	The Application Loader encountered an unexpected end of file on the object file represented by the command's pathname.
E$EXIST	At least one of the following is true: • The call detached the device containing the object file before completing the loading operation. • The command$conn parameter is not the token for a command connection.
E$FACCESS	The default user for the calling task's job does not have read access to the object file.
E$FIXUP	When the Application Loader loads an LTL (load-time-locatable) program, the Loader must adjust some of the addresses used in the code to reflect actual loaded code addresses. This adjustment is known as a fixup and is contained on a fixup record. The Application Loader detected an invalid fixup record in the object file. Refer to the iRMX 86 LOADER REFERENCE MANUAL for an explanation of LTL code.
E$FLUSHING	The device containing the object file was being detached.

E$FNEXIST	At least one of the following is true: • The file in the command's pathname is either marked for deletion or does not exist. • While attaching the file specified in the line$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
E$IO$HARD	While attempting to access the object file, this call detected a hard I/O error. This means that another try is probably useless.
E$IOMEM	The Basic I/O System does not currently have a block of memory large enough to allow the Human Interface to create the connection necessary to allow this call to run to completion.
E$IO$OPRINT	While attempting to access the object file, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value E$IO$NOT$READY when given this code.
E$IO$SOFT	While attempting to access the object file, this call detected a soft I/O error. It tried again, but was not successful. Another try might be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred while this call tried to access the object file.

E$IO$WRPROT	While attempting to obtain an input connection to the object file, the call found that the volume containing the file is write-protected.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job , or the job's default user object, is already involved in 255 (decimal) I/O operations. • The new I/O job, or its default user, is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E$LITERAL	The call found a literal (quoted string) with no closing quote while scanning the contents of the command line pointed to by the line$p parameter.
E$LOG$NAME$-NEXIST	The command's pathname contains an explicit logical name but the call was unable to find this name in the object directory of the local job, the global job, or the root job.
E$LOG$NAME$-SYNTAX	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	The device containing the object file was off-line.
E$MEM	The memory available to the calling task's job, the new I/O job, or the Basic I/O System job is not sufficient to complete the call.
E$NO$LOADER$MEM	At least one of the following is true: • The memory pool of the newly-created I/O job does not currently have a block of memory large enough to allow the Loader to run. • The memory pool of the Basic I/O System's job does not currently have a block of memory large enough to allow the Application Loader to run.

E$NO$MEM	The Application Loader attempted to load PIC or LTL groups or segments. However, the memory pool of the newly-created I/O job does not currently contain a block of memory large enough to accommodate these groups or segments. Refer to the iRMX 86 LOADER REFERENCE MANUAL for an explanation of loading PIC or LTL groups or segments.
E$NO$PREFIX	The calling task's job does not have a valid default prefix.
E$NO$START	The object file represented by the command-pathname does not specify the entry point for the program being loaded.
E$NOT$CONNECTION	The default$ci or default$co parameter is a token for an object that is not a file connection.
E$NOT$LOG$NAME	The command pathname contains a logical name. The logical name of an object that is neither a device connection nor a file connection.
E$NO$USER	The calling task's job does not have a valid default user.
E$PARAM	The Extended I/O System attempted the physical attachment of a device containing the object file. This device had formerly been only logically attached. While attempting this, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.
E$PARSE$TABLES	The call found an error in an internal table.
E$PATHNAME$-SYNTAX	The command's pathname contains invalid characters.
E$REC$FORMAT	At least one record in the object file contains a record format error.
E$REC$LENGTH	The object file contains a record that is longer than the Loader's maximum record length. The Loader's maximum record length is a parameter specified during the configuration of the Loader. Refer to the iRMX 86 CONFIGURATION GUIDE for details.
E$REC$TYPE	At least one of the following is true: • At least one record in the file being loaded is of a type that the Application Loader cannot process. • The Application Loader has encountered records in a sequence that it cannot process.

E$SEG$BOUNDS	The Application Loader created multiple segments in which to load information. One of the data records in the object file specified a load address outside of the created segments.
E$SEPARATOR	The call found an invalid separator while scanning the command line. The following is a list of the invalid command separators: ><, <>, ||, |, [, and].
E$STRING	The size of the command's pathname exceeds the length limit of 255 (decimal) characters.
E$STRING$BUFFER	The size of the command's pathname exceeds the size of the command name buffer specified during the configuration of the Human Interface.
E$TIME	The calling task's job was not created by the Human Interface.
E$TYPE	The command$conn parameter is token for an object that is not a command connection.

C$SEND$CO$RESPONSE

C$SEND$CO$RESPONSE, a message processing call, sends a message to :CO: and reads a response from :CI:.

```
CALL RQ$C$SEND$CO$RESPONSE(response$p, response$max, message$p,
                           except$ptr);
```

INPUT PARAMETERS

message$p	A POINTER to a STRING containing the message to be sent to :CO:. If zero, no message is sent.
response$max	A WORD that specifies the length in bytes of the string pointed to by the response$p parameter. If response$max is zero, no response from :CI: will be requested; control returns to the calling task immediately.

OUTPUT PARAMETERS

response$p	A POINTER to a STRING that receives the operator's response from :CI:.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

When used with all its features, C$SEND$CO$RESPONSE sends the string pointed to by message$p to :CO: and waits for a response from :CI:. It places this response in the string pointed to by response$p. However, If message$p is zero, C$SEND$CO$RESPONSE omits sending the message to :CO:; if either response$max or response$p is zero, it does not wait for a response from :CI:. Therefore, the operations performed by C$SEND$CO$RESPONSE depend on the values of the message$p and response$max parameters, as follows:

message$p	response$max	Action
zero	zero	Perform no I/O
zero	non-zero	Send no message, wait for input
non-zero	non-zero	Send message, wait for input
non-zero	zero	Send message, don't wait

If C$SEND$CO$RESPONSE requests a response from :CI:, output from other tasks can be displayed at :CO: while the system waits for a response from :CI:.

The main distinction between C$SEND$CO$RESPONSE and C$SEND$EO$RESPONSE calls is that C$SEND$EO$RESPONSE always sends messages to and receives messages from the operator's terminal; input and output cannot be redirected to another device. In contrast, C$SEND$CO$RESPONSE sends messages to :CO: and receives messages from :CI:; therefore, programs such as SUBMIT can redirect this input and output.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$CONTEXT	The calling task's job was not created by the Human Interface.
E$CONNECTION$-OPEN	At least one of the following is true: • The connection to :CI: was not open for reading or the connection to :CO: was not open for writing. • The connection to :CI: or :CO: was not open. • The connection to :CI: or :CO: was opened with A$OPEN rather than S$OPEN.
E$EXIST	The token value for :CI: or :CO: is not a token for an existing object.
E$FLUSHING	The device containing the :CI: and :CO: files was being detached.
E$IO$HARD	While attempting to access the :CI: or :CO: file, the Operating System detected a hard I/O error.
E$IO$OPRINT	While attempting to access the :CI: or :CO: file, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value E$IO$NOT$READY for this code.

E$IO$SOFT	While attempting to access the :CI: or :CO: file, this call detected a soft I/O error. It tried again, but was unsuccessful. Another try might be successful.
E$IO$UNCLASS	An unknown type of I/O error occurred while this call tried to access the :CI: or :CO: file.
E$IO$WRPROT	While attempting to obtain a connection to the :CO: file, this call found that the volume containing the file is write-protected.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$NOT$CONNECTION	The call obtained a token for an object that should have been a connection to :CI: or :CO: but was not a file connection.
E$PARAM	The call attempted to write beyond the end of a physical file.
E$SPACE	One of the following is true: • The output volume is full. • The call attempted to write beyond the end of a physical file.
E$STREAM$SPECIAL	When attempting to read or write to :CI: or :CO:, the Extended I/O System issued an invalid stream file request.
E$SUPPORT	The connection to :CI: or :CO: was not created by this job.
E$TIME	The calling task's job was not created by the Human Interface.

C$SEND$EO$RESPONSE

C$SEND$EO$RESPONSE, a message processing call, sends a message to and reads a response from the operator's terminal.

```
CALL RQ$C$SEND$EO$RESPONSE(response$p, response$max, message$p,
                           except$ptr);
```

INPUT PARAMETERS

message$p	A POINTER to a STRING containing the message to be sent to the operator's terminal. If zero, no message is sent.
response$max	A WORD that specifies the length in bytes of the string pointed to by the response$p parameter. If response$max is zero, no response from the operator's terminal will be requested; control returns to the calling task immediately.

OUTPUT PARAMETERS

response$p	A POINTER to a STRING that receives the operator's response from the terminal.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

When used with all its features, C$SEND$EO$RESPONSE sends the string pointed to by message$p to the operator's terminal and waits for a response from the operator. It places this response in the string pointed to by response$p. However, if message$p is zero, C$SEND$EO$RESPONSE omits sending the message to the operator; if either response$max or response$p is zero, it does not wait for a response. Therefore, the operations performed by C$SEND$EO$RESPONSE depend on the values of the message$p and response$max parameters, as follows:

message$p	response$max	Action
zero	zero	Perform no I/O
zero	non-zero	Send no message, wait for input
non-zero	non-zero	Send message, wait for input
non-zero	zero	Send message, don't wait

If C$SEND$EO$RESPONSE requests a response from the terminal, no other output can be displayed at the terminal until C$SEND$EO$RESPONSE receives a line terminator from the operator. However, the operator can choose to ignore the displayed message by entering a line terminator only.

The main distinction between C$SEND$CO$RESPONSE and C$SEND$EO$RESPONSE calls is that C$SEND$EO$RESPONSE always sends messages to and receives messages from the operator's terminal; input and output cannot be redirected to another device. In contrast, C$SEND$CO$RESPONSE sends messages to :CO: and receives messages from :CI:; therefore programs such as SUBMIT can redirect this input and output.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$CONNECTION$-OPEN	At least one of the following is true: • The connection to :CI: was not open for reading or the connection to :CO: was not open for writing. • The connection to :CI: or :CO: was not open. • The connection to :CI: or :CO: was opened with A$OPEN rather than S$OPEN.
E$CONTEXT	The calling task's job was not created by the Human Interface.
E$ERROR$-OUTPUT	Attempted to call SEND$EO$RESPONSE through an invalid method.
E$EXIST	The token value for :CI: or :CO: is not a token for an existing object.
E$FLUSHING	The device containing the :CI: and :CO: files was being detached.
E$IO$OPRINT	While attempting to access the terminal, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value E$IO$NOT$READY when given this code.

E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$MEM	The memory pool of the calling task's job does not currently have block of memory large enough to allow this system call to run to completion.
E$NOT$CONNECTION	The call obtained a token for an object that should have been a connection to :CI: or :CO: but was not a file connection.
E$PARAM	The call attempted to write beyond the end of a physical file.
E$STREAM$SPECIAL	When attempting to read or write to :CI: or :CO:, the Extended I/O System issued an invalid stream file request.
E$SUPPORT	The connection to the terminal was not created by this job.
E$TIME	The calling task's job was not created by the Human Interface.

C$SET$PARSE$BUFFER

C$SET$PARSE$BUFFER, a command parsing call, permits parsing the contents of a buffer other than the command line buffer whenever the parsing system calls are used.

```
offset = RQ$C$SET$PARSE$BUFFER(buff$p, buff$max, except$ptr);
```

INPUT PARAMETERS

buff$p	A POINTER to a buffer containing the text to be parsed. If the buff$p is zero, the buffer used for parsing reverts to the command line buffer and the buff$max parameter is ignored.
buff$max	A WORD that specifies the length in bytes of the string pointed to by the buff$p parameter.

OUTPUT PARAMETERS

offset	A WORD in which the Human Interface places the byte offset from the start of the parsing buffer of the last byte parsed in the previous parsing buffer.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

DESCRIPTION

C$SET$PARSE$BUFFER allows you to parse buffers other than the command line. You can change buffers at will; you can also revert to the command line parsing buffer by calling C$SET$PARSE$BUFFER with buff$p=0. However, only one parsing buffer per job can be active at any given time.

When called, C$SET$PARSE$BUFFER sets the parsing pointer to the beginning of the specified buffer. However, it also returns a value (in the offset parameter) that identifies the last byte parsed in the previous parsing buffer. This gives you the ability, when switching back to the previous buffer, of positioning the parsing pointer to its previous position with successive calls to C$GET$CHAR.

Note that C$SET$PARSE$BUFFER does not affect the buffer from which C$GET$INPUT$PATHNAME and C$GET$OUTPUT$PATHNAME retrieve pathnames. These system calls always obtain their pathnames from the command line.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$CONTEXT	The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • This indicates that the calling task's job was not created by the Human Interface.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.

***

CHAPTER 9 CONFIGURATION OF THE HUMAN INTERFACE

The Human Interface is a configurable part of the Operating System. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, Intel provides three kinds of information:

- A list of configurable options
- Detailed information about the options
- Procedures to allow you to specify your choices

The balance of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the iRMX 86 CONFIGURATION GUIDE.

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 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 86 directory structure and placing information about users into iRMX 86 files. The nonresident configuration information must be present when the application system starts running, but you can modify the information in the nonresident configuration files while the system is running. For the new nonresident configuration to take effect, you must reinitialize your application system.

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 8087 Numeric Processor Extension.
- Information about the initial user (or single user, if a single-access system), including terminal name, user ID, maximum priority, pathname of initial program, and default directory.

- 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.
- Whether the Human Interface includes an initial program that is linked to the Human Interface and used for all operators (resident initial program), or whether a separate initial program is used for each operator. If you include a resident initial program, you can also specify its pathname.

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 can specify:

- Terminal name
- Associated user name
- Memory partition size
- Maximum priority
- Pathname of the initial program

For each user in the system you can specify

- User ID
- Password
- Memory partition size
- Default prefix
- Pathname of the initial program
- Maximum job priority

***

APPENDIX A HUMAN INTERFACE TYPE DEFINITIONS

The type definitions used in Human Interface system call description are defined in Table A-1.

Table A-1. Type Definitions

Type	Definition
BYTE	An unsigned, eight-bit, binary number.
WORD	An unsigned, two-byte, binary number.
INTEGER	A signed, two-byte, binary number that is stored in two's complement form.
POINTER	Two consecutive words containing the base of a segment and the offset into that segment. The offset must be in the word having the lower address.
SELECTOR	A 16-bit quantity that is equivalent to the base portion of a pointer. Your PL/M compiler may not support this data type.
TOKEN	A word or selector whose value identifies an object. A TOKEN can be declared literally a WORD or a SELECTOR, depending on your needs.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes in the rest of the string. Since a string contains only a single byte in which to store the count, the maximum number of characters that a string can contain is 255. 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 HUMAN INTERFACE EXCEPTION CODES

Like other iRMX 86 software systems, the Human Interface returns a condition code whenever a Human Interface call is invoked. If the call executes without error, the Human Interface returns the code E$OK. When an error is encountered during call execution, an exceptional condition code is returned. The exceptional condition code may be returned either from the Human Interface or from one of the other iRMX 86 layers residing below it. The exception codes listed in Table B-1 are unique to the Human Interface.

Table B-1. Human Interface Exception Codes

Programmer Errors:	
E$PARSE$TABLES	8080H
E$JOB$TABLES	8081H
E$DEFAULT$SO	8083H
E$STRING	8084H
E$ERROR$OUTPUT	8085H
Environmental Errors:	
E$OK	0000H
E$LITERAL	0080H
E$STRING$BUFFER	0081H
E$SEPARATOR	0082H
E$CONTINUED	0083H
E$INVALID$NUMERIC	0084H
E$LIST	0085H
E$WILDCARD	0086H
E$PREPOSITION	0087H
E$PATH	0088H
E$CONTROL$C	0089H
E$CONTROL	008AH
E$UNMATCHED$LISTS	008BH
E$DATE	008CH
E$NO$PARAMETER	008DH
E$VERSION	008EH
E$GET$PATH$ORDER	008FH

The values of condition codes fall into ranges based on the iRMX 86 layer which first detects the condition. Table B-2 lists the layers and their respective ranges, with numeric values expressed in hexadecimal notation. Table B-3 lists all the exception codes for the operating system. All the exception codes are listed to their type (environmental errors, Nucleus programming errors, etc.). For more information on the exception codes, consult the manual which describes the layer from which the exception code originates.

Table B-2. Condition Code Ranges

Layer	Environmental Conditions	Programming Errors
Nucleus	0H 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

Note: * Exception codes in this range (130 to 14FH; 8130 to 814FH) could occur if you are a user of an iRMX system with iMMX 800 software. Refer to iMMX 800 MULTIBUS MESSAGE EXCHANGE REFERENCE MANUAL for an explanation of exception conditions within this range.

Table B-3. 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$EXIST	A token parameter has a value which is not the token of an existing object.	6H	6
E$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

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
	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$DEVFD	The device driver and file driver are incompatible.	22H	34
E$SUPPORT	The combination of parameters entered is not supported.	23H	35
E$EMPTY$-ENTRY	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
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

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Environmental Conditions (continued)			
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$-CONN	The connection parameter is not a file connection, but it should be.	32H	50
ENOTDEV-ICE$CONN	The connection parameter is not a device connection, but it should be.	33H	51
E$CONN$NOT$-OPEN	The connection is either closed or it is open for access not compatible with the current request.	34H	52
E$CONN$OPEN	The task attempted to open a connection that is already open.	35H	53
E$BUFFERED$-CONN	The specified connection was opened by the EIOS, which specified one or more buffers for the connection.	36H	54
E$OUTSTAND-ING$CONNS	A soft detach was specified, but connections to the device still exist.	37H	55
E$ALREADY$-ATTACHED	The specified device is already attached.	38H	56
EDEV-DETACHING	The file specified is on a device that the Operating System is detaching.	39H	57
E$NOT$SAME$-DEVICE	The existing pathname and the new path-name 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 di-rectory to a new path containing itself.	3BH	59

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Environmental Conditions (continued)			
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.	3F	63
E$LOG$NAME$-SYNTAX	The specified pathname starts with a colon (:), but it does not contain a second, matching colon.	40H	64
E$IOMEM	The Basic I/O System has insufficient memory to process a request.	42H	66
E$MEDIA	The device containing a specified file is not on-line.	44H	68
E$LOG$NAME$-NEXIST	The Extended I/O System was unable to find the specified logical name in the object directories that it checks.	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$IO$UNCLASS	An unknown type of I/O error occurred.	50H	80

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Environmental Conditions (continued)			
E$IO$SOFT	A soft I/O error occurred. A retry might be successful.	51H	81
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$MODE	A tape drive attempted a read (write) operation before the previous write (read) completed	56H	86
Application Loader Environmental Conditions			
E$BAD$GROUP	The group definition record contains an invalid group component.	61H	97
E$BAD$HEADER	The object file contains an invalid header record.	62H	98H
E$BAD$SEGDEF	The object file contains an invalid segment definition record.	63H	99H
E$CHECKSUM	A checksum error occurred while reading a record.	64H	100
E$EOF	The Application Loader encountered an unexpected end-of-file while reading a record.	65H	101
E$FIXUP	The file contains an invalid fixup record.	66H	102
E$NO$LOADER$-MEM	There is insufficient memory to satisfy the memory requirements of the Application Loader.	67H	103

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
Application Loader Environmental Conditions (continued)			
E$NO$MEM	There is insufficient memory to create PIC/LTL segments.	68H	104
E$REC$FORMAT	The file contains an invalid record format.	69H	105
E$REC$LENGTH	The record length exceeds the configured size of the Application Loader buffer.	6AH	106
E$REC$TYPE	The file contains an invalid record type.	6BH	107
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
E$OVERLAY	The overlay name does not match any of the overlay module names.	6EH	110
E$LOADER$-SUPPORT	The file requires features not supported by the Application Loader as configured.	6FH	111
E$SEG$BOUNDS	One of the data records in a module loaded by the Application Loader referred to an address outside the segment created for it.	70H	112
Human Interface Environmental Conditions			
E$LITERAL	The parsing buffer contains a literal with no closing quote.	80H	128
E$STRING$BUF-FER	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

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
Human Interface Environmental Conditions (continued)			
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$WILDCARD	A wild-card character appears in an invalid context, such as in an intermediate component of a pathname.	86H	134
E$PREPOSITION	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
E$UNMATCHED$-LISTS	The number of files in the input and output pathname lists is not the same.	8BH	139
E$DATE	The operator entered an invalid date.	8CH	140
E$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$PATHNAME	8FH	143
UDI Environmental Conditions			
E$UNKNOWN$EXIT	The program exited normally.	0C0H	192

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
	UDI Environmental Conditions (continued)		
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$UDI$INTERNAL	A UDI internal error occurred.	0C5H	197
	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 parameter 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$-BOUNDS	Hardware or software has detected an array overflow.	8006H	32774
* E$NDP$STATUS	A Numeric Processor Extension (NPX) error has occurred. OS extensions can return the status of the NPX to the exception handler.	8007H	32775
* E$ILLEGAL$-OPCODE	The iAPX 186 or 286 processor tried to execute an invalid instruction	8008H	32776

* For iAPX 286-based systems, a CPU trap caused this exceptional condition.

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
	Nucleus Programmer Errors (continued)		
* E$EMULATOR$-TRAP	The iAPX 186 or 286 processor tried to execute an ESC instruction with the "emulator" bit set in the relocation register (iAPX 186) or the machine status word (iAPX 286).	8009H	32777
* E$INTERRUPT$-TABLE$LIMIT	An iAPX 286 LIDT instruction changed the interrupt table limit to a value between 20H and 42H.	800AH	32778
* E$CPUXFER$-DATA$LIMIT	For an iAPX 286 processor, the processor extension data transfer exceeded the offset of 0FFFFH in a segment.	800BH	32779
* E$SEG$WRAP$-AROUND	For an iAPX 286 processor, either a word operation attempted a segment wraparound at offset 0FFFFH; or a PUSH, CALL, or INT instruction attempted to execute while SP = 1.	800CH	32780
E$CHECK$EX-CEPTION	A Pascal task has exceeded the bounds of a CASE statement.	8017H	32791
	I/O System Programmer Errors		
E$NOUSER	No default user is defined.	8021H	32801
E$NOPREFIX	No default prefix is defined.	8022H	32802
E$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

* For iAPX 286-based systems, a CPU trap caused this exceptional condition.

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Numeric Code Decimal
I/O System Programmer Errors (continued)			
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 memory pool size specified.	8060H	32864
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	32898
E$STRING	The pathname to be returned exceeds 255 characters in length.	8084H	32899
E$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$RESPONSE calls.	8085H	32900
UDI Programmer Errors			
E$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

***

APPENDIX C STRING TABLE FORMAT

The iRMX 86 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 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 C-1 shows the string$table parameter format.

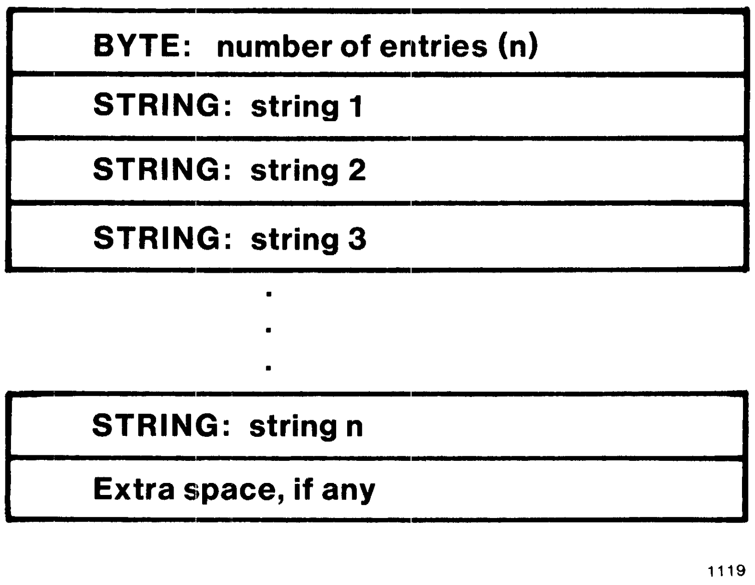


Figure C-1. String Table Format

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

Primary references are underscored.

***

iRMX™86 UNIVERSAL DEVELOPMENT INTERFACE REFERENCE MANUAL

CONTENTS

CONTENTS (continued)

***

CHAPTER 1 INTRODUCTION TO THE UNIVERSAL DEVELOPMENT INTERFACE

Intel's Universal Development Interface (known in this manual as the UDI) is a set of system calls that is compatible with each of Intel's operating systems. If an application system makes UDI system calls but no explicit calls to the resident 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 in between.

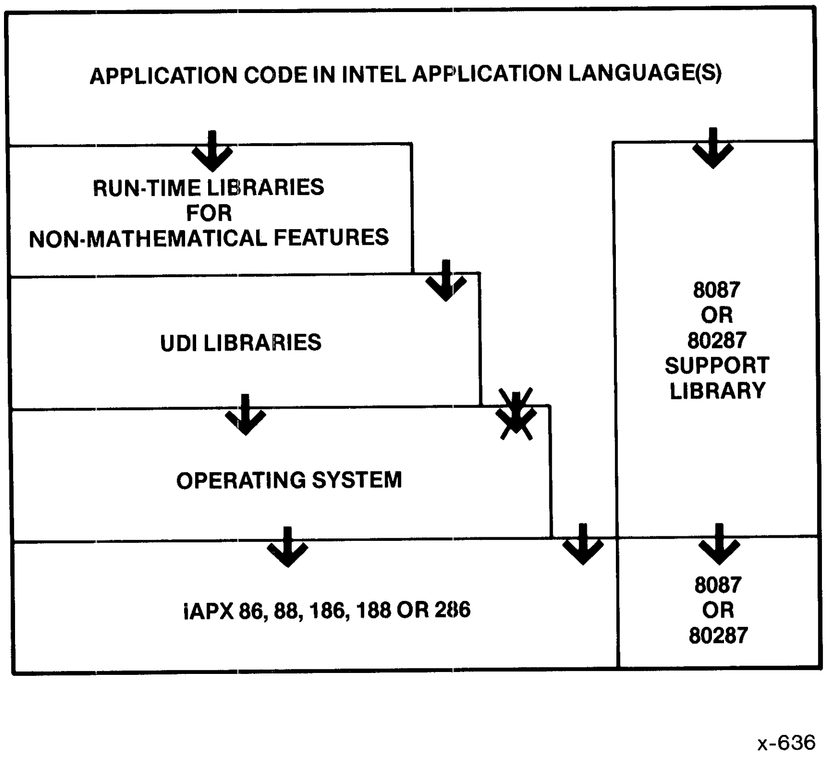

APPLICATION CODE IN INTEL APPLICATION LANGUAGE(S)
RUN-TIME LIBRARIES FOR NON-MATHEMATICAL FEATURES
UDI LIBRARIES
OPERATING SYSTEM
IAPX 86, 88, 186, 188 OR 286
8087 OR 80287 SUPPORT LIBRARY
8087 OR 80287
x-636

Figure 1-1. The Application-Software-Hardware Model

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.

By letting the UDI serve as the link between an application and the operating system, it is possible to switch operating systems simply by changing the interface between the UDI and the operating system. In other words, all that is necessary 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, iRMX 88, 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. The reason for this is that the operating systems each have many unique characteristics, and some of them are reflected in the results of the UDI calls. For information about the UDI and the minor behavioral differences it exhibits between operating systems, refer to the RUN-TIME SUPPORT MANUAL FOR iAPX 86,88 APPLICATIONS.

The next chapter discusses the UDI in the context of the iRMX 86 Operating System.

***

CHAPTER 2
UDI SYSTEM CALLS IN THE iRMX™ 86 ENVIRONMENT

The purpose of this chapter is to describe the requirements and behavior of UDI system calls in the iRMX 86 environment.

SYSTEM CALL DICTIONARY

This section presents, in Table 2-1, a list of the UDI calls, arranged by functional category. Each entry in the list includes the name of the call, a concise description of its purpose, and its page number in this chapter.

Table 2-1. System Call Dictionary

SYSTEM CALL	FUNCTION PERFORMED	PAGE
	PROGRAM-CONTROL CALLS	
DQ$EXIT	Exits from the current application job.	2-20
DQ$OVERLAY	Causes the specified overlay to be loaded.	2-37
DQ$TRAP$CC	Assigns Control-C procedure.	2-51
	MEMORY-MANAGEMENT CALLS	
DQ$ALLOCATE	Requests a memory segment of a specified size.	2-8
DQ$FREE	Returns a memory segment to the system.	2-25
DQ$GET$SIZE	Returns the size of a memory segment.	2-31
DQ$RESERVE$-IO$MEMORY	Requests memory to be set aside for overhead to be incurred by I/O operations.	2-42

Table 2-1. System Call Dictionary (continued)

SYSTEM CALL	FUNCTION PERFORMED	PAGE
FILE-HANDLING CALLS		
DQ$ATTACH	Creates a connection to a file.	2-9
DQ$CHANGE$-ACCESS	Changes the access rights associated with a file or directory.	2-10
DQ$CHANGE$-EXTENSION	Changes the extension of a file name.	2-12
DQ$CLOSE	Closes a file connection.	2-13
DQ$CREATE	Creates a file.	2-14
DQ$DELETE	Deletes a file.	2-18
DQ$DETACH	Closes a file and deletes a connection to it.	2-19
DQ$FILE$INFO	Returns data about a file connection.	2-22
DQGETCON-NECTION$STATUS	Returns the status of a file.	2-28
DQ$OPEN	Opens a file connection.	2-34
DQ$READ	Reads the next sequence of bytes from a file.	2-39
DQ$RENAME	Renames a file.	2-41
DQ$SEEK	Moves the current position pointer of a file.	2-44
DQ$SPECIAL	Sets the line-edit mode for a terminal.	2-46
DQ$TRUNCATE	Truncates a file to a specified length.	2-53
DQ$WRITE	Writes a sequence of bytes to a file.	2-54

Table 2-1. System Call Dictionary (continued)

SYSTEM CALL	FUNCTION PERFORMED	PAGE
EXCEPTION-HANDLING CALLS		
DQ$DECODE$-EXCEPTION	Converts an numeric exception code into its equivalent mnemonic.	2-15
DQGETEXCEPT-ION$HANDLER	Returns a POINTER to the current exception handler.	2-30
DQ$TRAP$-EXCEPTION	Identifies a custom exception handler to replace the current handler.	2-52
UTILITY CALLS		
DQ$DECODE$TIME	Returns system time and date in both binary and ASCII-character format.	2-16
DQ$GET$ARGUMENT	Returns an argument from a command line.	2-26
DQGET-SYSTEM$ID	Returns the name of the underlying operating system supporting the UDI.	2-37
DQ$GET$TIME	(Obsolete: included for compatability.)	2-33
DQ$SWITCH$BUFFER	Selects a new buffer to contain command lines.	2-49

OVERVIEW

This section discusses the functions of the many of the system calls, highlighting the interrelationships, if any, among the calls in the functional groups of Table 2-1.

MEMORY MANAGEMENT SYSTEM CALLS

When the iRMX 86 Operating System loads and runs a program, the program is allocated memory, in an amount that depends upon 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$ALLOCATE and DQ$FREE to get memory segments from the pool, and to return segments to the pool, respectively. They can also call DQ$GET$SIZE to receive information about allocated memory segments.

FILE-HANDLING SYSTEM CALLS

About one-half of UDI system calls are used to manipulate files. Figure 2-1 shows the chronological relationships among the most frequently used file-handling system calls.

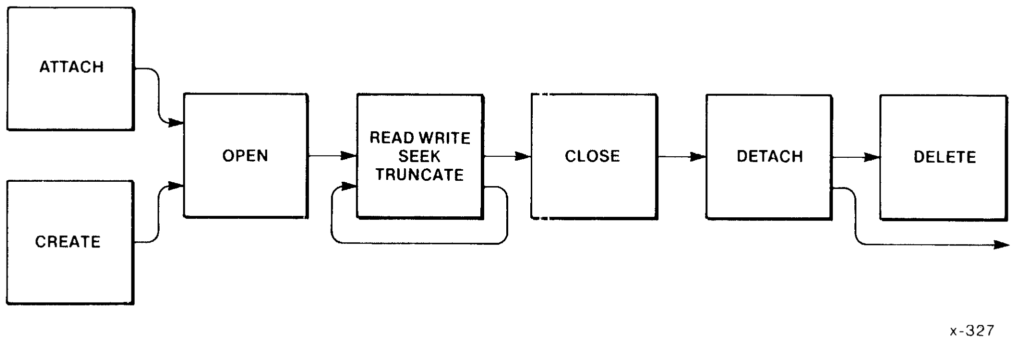


Figure 2-1. Chronology Of System Calls

The key to using iRMX 86 files is the <u>connection</u>. 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 has 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. 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 <u>connection</u>, not the file. Unless the program deletes the connection, 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 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 a file is marked for deletion, no new connections to it may be created.

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, so these conditions are called exceptions.

Exception conditions are classified as:

- <u>Environmental Conditions</u>. These are generally caused by conditions outside the control of a program; for example, device errors or insufficient memory.

- <u>Programmer Errors</u>. These are typically caused by mistakes in programming (for example, "bad parameter"), but "divide-by-zero", "overflow", "range check", and errors detected by the 8087 80287 Numeric Processor Extension (hereafter referred to generically as <u>the NPX</u>) are also classified as programmer errors.

The iRMX 86 NUCLEUS REFERENCE MANUAL contains a list of condition codes that the iRMX 86 Operating System can return, with the mnemonic and meaning of each code.

When an exception condition is detected, the normal (default) system action is to display an error message at the console and terminate the program. However, your program can establish its own exception handler by calling DQ$TRAP$EXCEPTION. The exception handler can interpret condition codes that are returned by calling DQ$DECODE$EXCEPTION. The rest of this section provides some facts that you need in order to write your own exception handler.

After an exception condition occurs and before your exception handler gets control, the iRMX 86 Operating System does the following:

1. Pushes the condition code onto the stack of the program that made the system call having the exception condition.

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, 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, but must use the default system exception handler. This is because alternate exception handlers must have a LONG POINTER, which is not available in the SMALL model.

MAKING UDI CALLS FROM PL/M-86 AND ASM86 PROGRAMS

This section describes how to make UDI calls from a program, using the DQ$ALLOCATE system call as an example. You can easily generalize from this example to see how to make the other UDI calls. There are two examples: one for a call from a PL/M-86 program and one for a call from an ASM86 program.

The way this chapter shows the DQ$ALLOCATE system call syntax is the following:

```
    base$addr = DQ$ALLOCATE (size, except$ptr);
```

There are three parameters: size (which has the WORD data type), except$ptr (which has the POINTER data type), and base$addr (which has WORD data type or the SELECTOR data type, depending on the version of PL/M-86).

Each of the examples that follow request 128 bytes of memory and point to a WORD named "ERR" where the condition code is to be returned.

EXAMPLE PL/M-86 CALLING SEQUENCE

```
    DECLARE    ARRAY_BASE    WORD, (or SELECTOR)
               ERR           WORD;
        •
        •
        •
        ARRAYBASE = DQ$ALLOCATE (128, @ERR);
```

EXAMPLE ASM86 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 assembled according to the COMPACT, MEDIUM, and LARGE models of segmentation. For the SMALL model, omit pushing the DS segment register.

DESCRIPTIONS OF SYSTEM CALLS

This section contains descriptions of the UDI system calls, which are arranged alphabetically. Every system call description contains the following information in this order:

- The name of the system call.
- A brief summary of the function of the call.
- The form of the call as it is invoked from a PL/M-86 program, with symbolic names for each parameter.
- Definition of input and output parameters.
- A complete explanation of the system call, including any information you will need to use the system call.

DQ$ALLOCATE

DQ$ALLOCATE requests a memory segment from the free memory pool.

```
    base$addr = DQ$ALLOCATE (size, except$ptr);
```

INPUT PARAMETER

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

OUTPUT PARAMETERS

base$addr	A SELECTOR, into which the Operating System places the base address of the memory segment. If the request fails because the memory requested is not available, this value will be OFFFFH, and the system will return an E$MEM exception code.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$ALLOCATE system call is used to request additional memory from the free space pool of the program. Tasks may use the additional memory for any desired purpose.

DQ$ATTACH

The DQ$ATTACH system call creates a connection to an existing file.

```
    connection = DQ$ATTACH (path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing the pathname of the file to be attached.

OUTPUT PARAMETERS

connection	A TOKEN for the connection to the file.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call allows a program to obtain a connection to any existing file. When the DQ$ATTACH call returns a connection, all existing connections to the file remain valid.

Your program can use the DQ$RESERVE$IO$MEMORY call to reserve memory that the UDI can use for its internal data structures when the program calls DQ$ATTACH and for buffers when the program calls DQ$OPEN. The advantage of reserving memory is that the memory is guaranteed to be available when needed. If memory is not reserved, a call to DQ$ATTACH might not be successful because of a memory shortage. See the description of DQ$RESERVE$IO$MEMORY later in this chapter for more information about reserving memory.

DQ$CHANGE$ACCESS

The DQ$CHANGE$ACCESS lets you change the access rights of the owner of a file (or directory), or the access rights of the WORLD user.

```
CALL DQ$CHANGE$ACCESS (path$ptr, user, access, except$ptr);
```

INPUT PARAMETERS

path$ptr — A POINTER to a STRING containing a pathname of the file.

user — A BYTE specifying the user whose access is to be changed:

Value	User
0	Owner of the file
1	WORLD (all users on the system)

access — A BYTE specifying the type of access to be granted the user. This word is to be encoded as follows. (Bit 0 is the low-order bit.)

Bit	Meaning
0	User can delete the file or directory
1	Read (the file) or List (the directory)
2	Append (the file) or Add entry (to the directory)
3	Update (read <u>and</u> write to the file) or Change Access (to the directory)
4-7	Should be zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

In the general iRMX 86 environment, every program is associated with a user object, usually referred to as the default user for the program. The default user consists of one or more user IDs. Each file has an associated collection of user ID-access mask pairs, where each mask defines the access rights the corresponding user ID has to the file. When the program calls DQ$CREATE to create a file or DQ$ATTACH to get another connection to a file, the resulting connection receives all access rights corresponding to user IDs that are both associated with the file and in the default user. The purpose of the DQ$CHANGE$ACCESS system call is to change, for a particular file, the access rights associated with a particular user ID. This has the effect of changing the access granted when the program makes subsequent calls to DQ$ATTACH to get further connections to the file.

In the UDI subset of the iRMX 86 environment, a default user has two IDs. One of them, called the owner ID, is associated with the program. The other, called the WORLD, is associated universally with all programs. DQ$CHANGE$ACCESS can change, for the file, the access mask of either the owner ID or the WORLD.

Changing the access rights for a user ID have no effect on connections already obtained by the program. However, all subsequently-obtained connections reflect the changed access rights.

For more information about user IDs, default users, access masks, WORLD, access rights, owner IDs, and how connections are related to all of these entities, refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

NOTE

DQ$CHANGE$ACCESS affects only connections made after the call is issued. It does not affect existing connections to the file.

DQ$CHANGE$EXTENSION

DQ$CHANGE$EXTENSION changes or adds the extension at the end of a file name stored in memory (not the file name on the mass storage volume).

```
    CALL DQ$CHANGE$EXTENSION (path$ptr, extension$ptr, except$ptr);
```

INPUT PARAMETERS

path$ptr	A POINTER to a STRING containing a pathname of the file to be renamed.
extension$ptr	A POINTER to a series of three bytes containing the characters to be added to the pathname. This is not a STRING. You must include three bytes, even if some are blank.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This is a facility for editing strings that represent file names in memory. If the existing file name has an extension, DQ$CHANGE$EXTENSION replaces that extension with the specified three characters. Otherwise, DQ$CHANGE$EXTENSION adds the three characters as an extension.

For example, a compiler can use DQ$CHANGE$EXTENSION to edit a string containing the name, such as :AFD1:FILE.SRC, of a source file to the name, such as :AFD1:FILE.OBJ, of an object file, and then create the object file.

Note that iRMX 86 file names may contain multiple periods, but if they do, the extension, if any, consists of the characters following the last period. Note also that an extension may contain more than three characters, but any extension created or changed by DQ$CHANGE$EXTENSION has at most three (non-blank) characters.

The three-character extension may not contain delimiters recognized by DQ$GET$ARGUMENT but may contain trailing blanks. If the first character pointed to by extension$ptr is a space, DQ$CHANGE$EXTENSION deletes the existing extension, if any, including the period preceding the extension.

DQ$CLOSE

DQ$CLOSE waits for completion of I/O operations (if any) taking place on the file, empties output buffers, and frees all buffers associated with the connection.

```
    CALL DQ$CLOSE (connection, except$ptr);
```

INPUT PARAMETER

connection	A TOKEN for a file connection that is currently open.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$CLOSE system call closes a connection that has been opened by the DQ$OPEN system call. It performs the following actions, in order:

1. Waits until all currently-running I/O operations for the connection are completed.

2. Ensures that information, if any, in a partially-filled output buffer is written to the file.

3. Releases all buffers associated with the connection.

4. Closes the connection. The connection is still valid, and can be re-opened if necessary.

DQ$CREATE

DQ$CREATE creates a new file and establishes a connection to the file.

```
    connection = DQ$CREATE (path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing a pathname of the file to be created.

OUTPUT PARAMETERS

connection	A TOKEN for the connection to the file.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This call creates a new file with the name you specify and returns a connection to it. If a file of the same name already exists, it is truncated to zero length and the data in it is destroyed.

To prevent accidentally destroying a file, call DQ$ATTACH before calling DQ$CREATE. If the file does <u>not</u> exist, DQ$ATTACH returns an E$FNEXIST exception code.

DQ$DECODE$EXCEPTION

DQ$DECODE$EXCEPTION translates an exception code into its mnemonic.

```
    CALL DQ$DECODE$EXCEPTION (except$code, buff$ptr, except$ptr);
```

INPUT PARAMETER

except$code	A WORD containing the numeric exception code that is to be translated.

OUTPUT PARAMETERS

buff$ptr	A POINTER to a buffer (at least 81 bytes long) into which the system returns the mnemonic in a STRING.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Your program can call DQ$DECODE$EXCEPTION to exchange a numeric exception code for its hexadecimal equivalent followed by its mnemonic. For example, if you pass DQ$DECODE$EXCEPTION a value of 2 in the except$code parameter, the system returns the following string to the area pointed to by the buff$ptr parameter:

```
    0002: E$MEM
```

The hexadecimal values and mnemonics for condition codes are listed in Appendix B.

DQ$DECODE$TIME

DQ$DECODE$TIME returns the current system time and date as a Double Word integer and as a series of ASCII character bytes.

```
    CALL  DQ$DECODE$TIME (time$ptr, except$ptr);
```

OUTPUT PARAMETERS

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

```
DECLARE DT STRUCTURE(
    SYSTEM$TIME         DWORD,
    DATE (8)            BYTE,
    TIME (8)            BYTE);
```

If the value in SYSTEM$TIME is 0 when DQ$DECODE$TIME is called, DQ$DECODE$TIME returns the current date and time in the DT structure, as follows. (See the following DESCRIPTION section for format information.):

SYSTEM$TIME receives the time as the number of seconds since midnight, January 1, 1978.

DATE receives the date portion of the time, in the form of ASCII characters.

TIME receives the time-of-day portion of the time, in the form of ASCII characters.

If the value in SYSTEM$TIME is not 0 when DQ$DECODE$TIME is called, DQ$DECODE$TIME accepts that value as the number of seconds since midnight, January 1, 1978, decodes the value, and returns it in the DATE and TIME fields.

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call returns the indicated date and time, each as a series of ASCII bytes. (Note that they are not STRINGs.)

DATE has the form MM/DD/YY for month, day, and year. The two slashes (/) are in the third and sixth bytes. For example, the date January 15th of 1982 would be returned as:

```
01/15/82
```

TIME has the form HH:MM:SS for hours, minutes, and seconds, with separating colons (:). The value for hours ranges from 0 through 23. For example, the time 20 seconds past 3:12 PM would be returned as:

```
15:12:20
```

If, when you call DQ$DECODE$TIME, the SYSTEM$TIME parameter is zero, the call first gets the system time (number of seconds since midnight, January 1, 1978) and then decodes it into the series of bytes as just described.

But if SYSTEM$TIME is not zero on input, DQ$DECODE$TIME uses it as the time to decode.

One thing your program can do with DQ$DECODE$TIME is first to call DQ$FILE$INFO to get two DWORD values associated with a file (the last time the file was updated and the time the file was created). Then the program can call DQ$DECODE$TIME to interpret the times.

DQ$DELETE

DQ$DELETE deletes an existing file.

```
    CALL DQ$DELETE (path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing a pathname of the file to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

A program can use this system call to delete a file. The immediate action this call takes is to mark the file for deletion. It does this rather than abruptly deleting the file, because it will not delete any file as long as there are existing connections to the file. DQ$DELETE will delete the file only when there are no longer any connections to the file, that is, when all existing connections have been detached. On the other hand, once the file is marked for deletion, no more connections may be obtained for the file by way of DQ$ATTACH.

DQ$DETACH

DQ$DETACH deletes a connection (but not the file) established by DQ$ATTACH or DQ$CREATE.

```
    CALL DQ$DETACH (connection, except$ptr);
```

INPUT PARAMETER

connection	A TOKEN for the file connection to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call deletes a file connection. If the connection is open, the DQ$DETACH system call automatically closes it first (see DQ$CLOSE). DQ$DETACH also deletes the file if the file has been marked for deletion and this is the last existing connection to the file.

DQ$EXIT

DQ$EXIT transfers control from your program to the iRMX 86 Operating System. It does not return any value to the calling program, not even a condition code.

```
    CALL DQ$EXIT (end$code);
```

INPUT PARAMETERS

end$code	A WORD containing the encoded reason for termination of the program. See the following description for information about this value.

DESCRIPTION

DQ$EXIT terminates a program. Before the actual termination, all of the program's connections are closed and detached, and all memory allocated to the program by DQ$ALLOCATE is returned to the memory pool.

DQ$EXIT does not return a condition code to the calling program.

If the calling program is running as an I/O job, the calling task, normally the command line interpreter (CLI), receives an iRMX 86 condition code based on the value your program supplied in the end$code field when it called DQ$EXIT. This assumes the following sequence of events:

1. The CLI calls RQ$CREATE$IO$JOB, specifying a response mailbox in the call.

2. Your program, running as a task in the created I/O job, performs its duties and then calls DQ$EXIT, specifying an end$code value.

3. DQ$EXIT converts the end$code value into an iRMX 86 condition code, as follows:

end$code Value	iRMX 86 Condition Code	Associated Mnemonic	Meaning
0	0C0H	E$UNKNOWN$EXIT	Termination was normal.
1	0C1H	E$WARNING$EXIT	Warning messages were issued.
2	0C2H	E$ERROR$EXIT	Errors were detected.
3	0C3H	E$FATAL$EXIT	Fatal errors were detected.
4	0C4H	E$ABORT$EXIT	The job was aborted.
5-65535	0C0H	E$UNKNOWN$EXIT	Cause of termination not known.

4. DQ$EXIT calls RQ$EXIT$IO$JOB, specifying the iRMX 86 condition code in the user$fault$code field.

5. RQ$EXIT$IO$JOB places the condition code into the user$fault$code field of a message. Then RQ$EXIT$IO$JOB sends the message to the response mailbox set up by the earlier call to RQ$CREATE$IO$JOB.

6. The CLI, when it obtains the message from the response mailbox, can take appropriate actions. Note that it can call DQ$DECODE$EXCEPTION first, to convert the condition code into its associated mnemonic.

The CLI program supplied with the iRMX 86 Operating System ignores these UDI condition codes when they are returned in the user$fault$code field of the response message. Therefore, if you want the CLI to take actions based on that code, you must provide your own CLI.

For more information about RQ$CREATE$IO$JOB, RQ$EXIT$IO$JOB, and the format of the response message, see the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

DQ$FILE$INFO

DQ$FILE$INFO returns information about a file.

```
CALL  DQ$FILE$INFO (connection, mode, file$info$ptr, except$ptr);
```

INPUT PARAMETERS

connection	A TOKEN containing a connection for the file.
mode	An encoded BYTE specifying whether DQ$FILE$INFO is to return the User ID of the owner of the file. Encode as follows:

Value	Meaning
0	Do not return owner's User ID.
1	Return the owner's User ID.

OUTPUT PARAMETERS

file$info$ptr — A POINTER to a structure into which the requested information is to be returned. The form of the structure is:

```
DECLARE FDATA STRUCTURE(
    OWNER(15)          STRING,
    LENGTH             DWORD,
    TYPE               BYTE,
    OWNER$ACCESS       BYTE,
    WORLD$ACCESS       BYTE,
    CREATE$TIME        DWORD,
    LAST$MOD$TIME      DWORD,
    RESERVED(20)       BYTE);
```

where:

OWNER	A STRING containing (if requested) the User ID of the file owner.
TYPE	A value indicating the type of file, as follows:

Value	File Type
0	Data file
1	Directory file

OWNER$ACCESS	An encoded BYTE whose bits specify the type of access granted to the owner, as follows. When a bit is set, it means the type of access is granted; otherwise the type of access is denied. (Bit 0 is the low-order bit.)

Bit	Associated Type of Access
0	Delete
1	Read (the data file) or Display (the directory)
2	Append (to the data file) or Add Entry (to the directory)
3	Update (read <u>and</u> write to the file) or Change Access (to the directory)

WORLD$ACCESS	An encoded BYTE whose bits specify the type of access granted to the WORLD (all users on the system). When a bit is set, it means the type of access is granted; otherwise the type of access is denied. (Bit 0 is the low-order bit.)

Bit	Associated Type of Access
0	Delete
1	Read (the data file) or Display (the directory)
2	Write (to the data file) or Add Entry (to the directory)
3	Update (read <u>and</u> write to the file) or Change Access (to the directory)

CREATE$TIME	The date and time that the file was created, expressed as the number of seconds since midnight, January 1, 1978. (You can convert this date/time to ASCII characters by calling DQ$DECODE$TIME.)

LAST$MOD$TIME	The date and time that the file or directory was last modified. For data files, modified means written or truncated; for directories, modified means an entry was changed or an entry was added. (You can convert this date/time to ASCII characters by calling DQ$DECODE$TIME.)

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$FILE$INFO returns information about a data file or a directory file.

DQ$FREE

DQ$FREE returns to the system a segment of memory obtained earlier by DQ$ALLOCATE.

```
    CALL DQ$FREE (base$addr, except$ptr);
```

INPUT PARAMETER

base$addr	A TOKEN containing the base address of the segment to be deleted. This value is the token returned by DQ$ALLOCATE when the segment was obtained.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$FREE system call returns the specified segment to the memory pool from which it was allocated.

DQ$GET$ARGUMENT

The DQ$GET$ARGUMENT system call returns arguments, one at a time, from a command line entered at the system console. This command line is either that which invoked the program containing the DQ$GET$ARGUMENT call or a command line entered while the program was running.

```
    delimit$char =  DQ$GET$ARGUMENT (argument$ptr, except$ptr);
```

INPUT PARAMETER

argument$ptr	A POINTER to a buffer which will receive the argument in the form of a STRING. The buffer must be at least 81 bytes long.

OUTPUT PARAMETERS

delimit$char	A BYTE which receives the delimiter character.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Your program can call GET$ARGUMENT to get arguments from a command line. Each call returns an argument and the delimiter character following the argument.

Your program can use this command in two ways. One way is to get arguments from the command line used to invoke the program at the console. In this case, you can assume that the command line is already in a buffer that has automatically been provided for this purpose.

The other way to use this command is to get arguments from command lines that are entered in response to requests from your program. In this case, your program must supply a buffer when calling DQ$READ, so this is the buffer you want to be used when your program calls DQ$GET$ARGUMENT. To set this up, your program must call DQ$SWITCH$BUFFER before the call to DQ$GET$ARGUMENT.

A delimiter is returned only if the exception code is zero. The following delimiters are recognized by the iRMX 86 Operating System:

,) (= # ! $ % \ + - > < ~

as well as a space ( ) and all characters with ASCII values in the range 0 through 20H.

Before returning arguments in response to DQ$GET$ARGUMENT, the system does the following editing on the contents of the command buffer:

- It strips out ampersands (&) and semicolons (;).
- Where multiple blanks are adjacent to each other between arguments, it replaces them with a single blank. (Tabs are treated as blanks.)
- It converts lowercase characters to uppercase unless they are part of a quoted string.

When returning arguments in response to DQ$GET$ARGUMENT, the system considers strings enclosed between matching pairs of single or double quotes to be literals. The enclosing quotes are not returned as part of the argument.

EXAMPLE

The following example illustrates the arguments and delimiters returned by successive calls to DQ$GET$ARGUMENT. The example assumes that the contents of the buffer are

```
PLM86 LINKER.PLM PRINT(:LP:) NOLIST
```

The following shows what is returned in this case if DQ$GET$ARGUMENT is called five times.

CALL NUMBER	ARGUMENT RETURNED	DELIMITER RETURNED
1	(05H)PLM86	space
2	(0AH)LINKER.PLM	space
3	(05H)PRINT	(
4	(04H):LP:	)
5	(06H)NOLIST	cr

Note that the argument returned has the form of an iRMX 86 string, with the first byte devoted to specifying the length of the string. In the second call, there are ten characters in the argument, so the first byte contains 0AH.

Note that the last delimiter for each example is a carriage return (cr). This is how your program can determine that there are no more arguments in the command line.

DQ$GET$CONNECTION$STATUS

The DQ$GET$CONNECTION$STATUS system call returns information about a file connection.

```
CALL DQ$GET$CONNECTION$STATUS (connection, info$ptr, except$ptr);
```

INPUT PARAMETER

connection	A WORD containing a token for the connection whose status is desired.

OUTPUT PARAMETERS

info$ptr — A POINTER to a structure into which the Operating System is to place the status information. The structure has the following format:

```
DECLARE INFO STRUCTURE(
     OPEN           BYTE,
     ACCESS         BYTE,
     SEEK           BYTE,
     FILE$PTR$      DWORD);
```

Where:

OPEN	1 if the connection is open; 2 otherwise.
ACCESS	Access privileges of the connection. The right is granted if the corresponding bit is set to 1. (Bit 0 is the low-order bit.)

Bit	Access
0	Delete
1	Read
2	Write
3	Update (read and write)

SEEK — Types of seek supported.

Value	Meaning
0	No seek allowed
3	Seek forward and backward

Other values are not meaningful.

FILE$PTR — This DWORD integer marks the current position in the file. The position is expressed as the number of bytes from the beginning of the file, the first byte being byte 0. This field is undefined if the file is not open or if seek is not supported by the device. (For example, seek operations are not valid for a line printer.)

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

DQ$GET$CONNECTION$STATUS returns information about a file CONNECTION. You might use this system call, for example, if your program has performed several read or write operations and it is necessary to determine where the file pointer is now located.

DQ$GET$EXCEPTION$HANDLER

DQ$GET$EXCEPTION$HANDLER returns the address of the current exception handler.

```
    CALL DQ$GET$EXCEPTION (address$ptr, except$ptr);
```

OUTPUT PARAMETERS

address$ptr	A POINTER to a POINTER into which this system call returns the entry point of the current exception handler.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

DQ$GET$EXCEPTION$HANDLER is an system call that returns to your program the address of the current exception handler. This is the address specified in the most recent call, if any, to DQ$TRAP$EXCEPTION. Otherwise the value returned is the address of the system default exception handler.

This routine always returns a two-word pointer, even if called from a program compiled under the SMALL model of segmentation.

DQ$GET$EXCEPTION$HANDLER is used in conjunction with DQ$TRAP$EXCEPTION and DQ$DECODE$EXCEPTION. See the descriptions of these calls for more information.

DQ$GET$SIZE

DQ$GET$SIZE returns the size of a previously-allocated memory segment.

```
    size = DQ$GET$SIZE (base$addr, except$ptr);
```

INPUT PARAMETER

base$addr	A TOKEN for a segment of memory that has been allocated by the DQ$ALLOCATE call. This is the same address returned by DQ$ALLOCATE when the segment was allocated.

OUTPUT PARAMETERS

size	A WORD which, if not zero, contains the size, in bytes, of the segment identified by the base$addr parameter. if zero, indicates that the size of the segment is 65536 (64K) bytes.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The GET$SIZE system call returns the size, in bytes, of a segment. The size of the segment might not be exactly what was originally requested for the segment, because DQ$ALLOCATE allocates memory in 16-byte paragraphs. If a request is for a size that is not a multiple of 16, DQ$ALLOCATE increases the size of the request to the next higher multiple of 16 before acting upon the request.

DQ$GET$SYSTEM$ID

DQ$GET$SYSTEM$ID returns the identity of the operating system providing the environment for the UDI.

```
    CALL DQ$GET$SYSTEM$ID (id$ptr, except$ptr);
```

OUTPUT PARAMETERS

id$ptr	A POINTER to a 21-byte buffer into which DQ$GET$SYSTEM$ID places a STRING identifying the operating system.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call returns the string:

    iRMX 86

followed by 13 blanks.

DQ$GET$TIME

DQ$GET$TIME returns the current date and time in character format.

```
    CALL DQ$GET$TIME (buff$ptr, except$ptr);
```

This system call performs no action except that it returns. It is included only for compatibility with previous versions of the UDI. You should use the DQ$DECODE$TIME system call for this function.

DQ$OPEN

The DQ$OPEN system call opens a file for I/O operations, specifies how the file will be accessed, and specifies the number of buffers needed to support the I/O operations.

```
    CALL DQ$OPEN (connection, access, num$buf, except$ptr);
```

INPUT PARAMETERS

connection	A TOKEN for the file connection to be opened.
access	A BYTE specifying how the connection will be used to access the file. This value is encoded as follows:

Value	Meaning
1	Read only
2	Write only
3	Update (both reading and writing)

num$buf	A BYTE containing the number of buffers needed for this connection. Specifying a value larger than 0 implicitly requests that "double buffering" (that is, read-ahead and/or write-behind) is to be performed automatically.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call prepares a connection for use with DQ$READ, DQ$WRITE, DQ$SEEK, and DQ$TRUNCATE commands. Any number of connections to the same file may be open simultaneously.

The DQ$OPEN system call does the following:

- Creates the requested buffers.

- Sets the connection's file pointer to zero. This a place marker that tells where in the file the next I/O operation is to begin.

- Starts reading ahead if num$buf is greater than zero and the access parameter is "Read only" or "Update."

Selecting Access Rights

The system does not allow reading using a connection open for writing only nor writing using a connection open for reading only. If you are not certain how the connection will be used, specify updating. However, if the specified connection does not support the specified type of access, an exception code is returned.

Selecting the Number of Buffers

The process of deciding how many buffers to request is based on three considerations -- compatibility, memory, and performance.

COMPATIBILITY. If you expect to run your UDI program on other systems, you should request no more than two buffers.

MEMORY. The amount of memory used for buffers is directly proportional to the number of buffers. So you can save memory by using fewer buffers.

PERFORMANCE. The performance consideration is more complex. Up to a certain point, the more buffers you allocate, the faster your program can run. The actual break-even point, where more buffers don't improve performance, depends on many variables. Often, the only way to determine the break-even point is to experiment. However, the following statements are true of every system:

- To overlap I/O with computation, you must request at least two buffers.

- If performance is not at all important but memory is, request no buffers.

Requesting zero buffers means that no buffering is to occur. That is, each DQ$READ or DQ$WRITE is followed immediately by the physical I/O operation necessary to perform the requested reading or writing. Interactive programs should open :CI: and :CO: with a request for no buffers.

If your program normally calls DQ$SEEK before calling DQ$READ or DQ$WRITE, it should request one buffer.

Your program can use the DQ$RESERVE$IO$MEMORY call to reserve memory that the UDI can use for its internal data structures when the program calls DQ$ATTACH and for buffers when the program calls DQ$OPEN. The advantage of reserving memory is that the memory is guaranteed to be available when needed. If memory is not reserved, a call to DQ$OPEN might not be successful because of a memory shortage. See the description of DQ$RESERVE$IO$MEMORY later in this chapter for more information about reserving memory.

DQ$OVERLAY

In systems using overlays, the root module calls DQ$OVERLAY to load an overlay module.

```
    CALL DQ$OVERLAY (name$ptr, except$ptr);
```

INPUT PARAMETER

name$ptr	A POINTER to a STRING containing the name of an overlay module. The name must be in uppercase.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

A root module in an overlay system calls DQ$OVERLAY each time it wants to load an overlay module.

If your assembly language or PL/M-86 program uses the DQ$OVERLAY procedure, you should take care to ensure that you link the UDI library to your program correctly. The iAPX 86, 88 FAMILY UTILITIES USER'S GUIDE contains an example of linking an overlay program. This example lists a two-step link process, as follows:

1. Link the root and each of the overlays separately, specifying the OVERLAY control, but not the BIND control, in each LINK86 command.

2. Link all the output modules together in one module, specifying the BIND control, but not the OVERLAY control.

This is the same process you should use when linking your iRMX 86 overlay programs.

In addition, you must link the entire UDI library to the root portion of the program and not to any of the overlays. To do this, use the INCLUDE control to include the UDI externals file when assembling or compiling the root portion of the program. By including this file with the root module, you ensure that the root module makes external references to all UDI routines. This prevents unsatisfied external references when the root is linked to the overlays.

DQ$READ

The DQ$READ system call copies bytes from a file into a buffer.

```
    bytes$read = DQ$READ (connection, buff$ptr, bytes$max,
                          except$ptr);
```

INPUT PARAMETERS

connection	A TOKEN for the connection to the file. This connection must be open for reading or for both reading and writing, and the file pointer of the connection must point to the first byte to be read.
buff$ptr	A POINTER to the buffer that is to receive the data from the file.
bytes$max	A WORD containing the maximum number of bytes to be read from the file.

OUTPUT PARAMETERS

bytes$read	A WORD containing the number of bytes actually read. This number is always equal to or less than the bytes$max.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call reads a collection of contiguous bytes from the file associated with the connection. The bytes are placed into the buffer specified in the call.

The Buffer

The buff$ptr parameter tells the Operating System where to place the bytes when they are read. Your program must provide this buffer. DQ$READ copies as many bytes as it is instructed to copy (unless it encounters the end of the file), so if the buffer is not long enough, copying continues beyond the end of the buffer.

Number of Bytes Read

The number of bytes that your program requests is the maximum number of bytes that DQ$READ copies into the buffer. However, there are two circumstances under which the system reads fewer bytes.

- If the DQ$READ detects an end of file before reading the number of bytes requested, it returns only the bytes preceding the end of file. In this case, the bytes$read parameter is less than the bytes$desired parameter, yet no exceptional condition is indicated.
- If an exceptional condition occurs during the reading operation, information in the buffer and the value of the bytes$read parameter are meaningless and should be ignored.

Connection Requirements

The connection must be open for reading or updating. If it is not, DQ$READ returns an exceptional condition.

DQ$RENAME

The DQ$RENAME system call changes the pathname of a file.

```
    CALL DQ$RENAME (path$ptr, new$path$ptr, except$ptr);
```

INPUT PARAMETERS

path$ptr	A POINTER to a STRING that specifies the pathname for the file to be renamed.
new$path$ptr	A POINTER to a STRING that specifies the new pathname for the file. This path must not refer to an existing file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call allows your programs to change the pathname of a data file or a directory. Be aware that when you rename a directory, you are changing the pathnames of all files contained in the directory. When you rename a file to which a connection exists -- this is permitted -- the connection to the renamed file remains established.

A file's pathname may be changed in any way, provided that the file or directory remains on the same volume.

DQ$RESERVE$IO$MEMORY

The DQ$RESERVE$IO$MEMORY lets your program reserve enough memory to ensure that it can open and attach the files it will be using.

```
    CALL DQ$RESERVE$IO$MEMORY (number$files, number$buffers, except$ptr);
```

INPUT PARAMETERS

number$files	The maximum number of files the program will have attached simultaneously. This value must not be greater than 12. Moreover, no more than 6 of these files may be open simultaneously.
number$buffers	The total number of buffers (up to a maximum of 12) that will be needed at one time. For example, if your program will have two files open at the same time, and each of them has two buffers (specified when they are opened), number$files should be two and number$buffers four.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

DQ$RESERVE$IO$MEMORY sets aside memory on behalf of the calling program, guaranteeing that it will be available when needed later for attaching and opening files. This memory is used for internal UDI data structures when the program requests file connections via DQ$ATTACH and for buffers when the program opens file connections via DQ$OPEN. Memory reserved in this way is not eligible to be allocated by DQ$ALLOCATE. Your program should call DQ$RESERVE$IO$MEMORY before making any calls to DQ$ALLOCATE.

In the call to DQ$RESERVE$IO$MEMORY, you may specify as many as 12 files (that can be attached using the reserved memory) and as many as 12 buffers (that can be requested when opening files).

NOTE

> If a program calls DQ$RESERVE$IO$MEMORY after making one or more calls to DQ$ATTACH or DQ$OPEN, the memory used by those calls are immediately applied against the file and buffer counts specified in the DQ$RESERVE$IO$MEMORY call, possibly exhausting the memory supply being requested.

If your program calls DQ$RESERVE$IO$MEMORY more than once in a program, it simply changes the amount of memory reserved.

RESTRICTION

This system call is effective only if your program uses exclusively UDI system calls to communicate with the iRMX 86 Operating System.

DQ$SEEK

DQ$SEEK moves the file pointer associated with the specified connection.

```
CALL DQ$SEEK (connection, mode, move$count, except$ptr)
```

INPUT PARAMETERS

connection	A TOKEN for the open connection whose file pointer is to be moved.
mode	A BYTE indicating the type of file pointer movement being requested, as follows:

Mode	Meaning
1	Move the pointer backward by the specified move count. If the move count is large enough to position the pointer past the beginning of the file, set the pointer to the first byte (position zero).
2	Set the pointer to the position specified by the move count. Position zero is the first position in the file. Moving the pointer beyond the end of the file is permitted.
3	Move the file pointer forward by the specified move count. Moving the pointer beyond the end of the file is permitted.
4	First move the pointer to the end of the file and then move it backward by the specified move count. If the specified move count would position the pointer beyond the front of the file, set the pointer to the first byte in the file (position zero).

move$count	A DWORD specifying how far, in bytes, the file pointer is to be moved.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

When performing non-sequential I/O, your programs can use this system call to position the file pointer before using the DQ$READ, DQ$TRUNCATE, or DQ$WRITE system calls. The location of the file pointer specifies where in the file a DQ$READ, DQ$WRITE, or DQ$TRUNCATE operation is to begin. If your program is performing sequential I/O on a file, it need not use this system call.

It is legitimate to position the file pointer beyond the end of a file. If your program does this and then invokes the DQ$READ system call, DQ$READ behaves as though the read operation began at the end of file. If your program calls DQ$WRITE when the file pointer is beyond the end of the file, the data is written as requested. Be aware that if you expand your file in this manner, the expanded portion of the file can contain undefined information.

DQ$SPECIAL

DQ$SPECIAL specifies whether line editing features are to be available to operators entering information at the console.

```
    CALL DQ$SPECIAL (mode, conn$ptr, except$ptr);
```

INPUT PARAMETERS

mode — A BYTE used to specify the mode of terminal input. The values and their meanings are:

Value	Meaning
1	Transparent
2	Line editing
3	Immediate transparent

Each of these types is explained in the DESCRIPTION section.

conn$ptr — A POINTER to a TOKEN for a connection to the :CI: file. The connection must have been established by DQ$ATTACH.

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call changes the mode in which your program receives input from a console input device. When your system starts to run, the mode is line editing (mode 2). But by using DQ$SPECIAL you can change from line editing to one of the transparent modes, or back to line editing.

The Line Editing Modes

The meanings of the mode parameter are as follows:

Value	Meaning
1	Transparent. Interactive programs often need to obtain characters from the console exactly as they are typed. This is made possible by transparent mode. In transparent mode, all characters are placed in the buffer specified by the call to DQ$READ. (The only exceptions are CTRL/C, which terminates the program, and CTRL/D, which is discarded.) DQ$READ returns control to the calling program when the number of characters entered equals the number of characters specified in the read request.
2	Line Editing. This option means that the console operator has the opportunity to correct typing errors with special keys before the application program receives the characters typed. Line editing characters and their effects are described following the descriptions of these line editing modes.
3	Immediate Transparent. This option is nearly the same as Transparent 1 mode, except that in Transparent 3 mode DQ$READ returns control to your program immediately after it is called, regardless of whether any characters have been typed since the last call to DQ$READ. If no characters have been typed, this is indicated by the bytes$read parameter of the DQ$READ call. Characters that are typed between successive calls to read the terminal are held in the "type-ahead" buffer.

The Line Editing Characters

The following characters and control characters have the following special editing capabilities on console input when line editing mode (mode 2) is in effect:

CARRIAGE RETURN or LINE FEED	Terminates the current line and positions the cursor at the beginning of the next line. Entering either of these characters adds a carriage return/line feed pair to the input line.
RUBOUT	Deletes (rubs out) the previous character in the input line. Each RUBOUT removes a character from both the screen and the type-ahead buffer, and moves the cursor back to that character position.

CTRL/R	If the current input line is not empty, this character reprints the line with editing already performed. This enables the operator to see the effects of the editing performed since the most recent line terminator was entered. If the current line is empty, CTRL/R reprints the previous line. Additional CTRL/Rs display previous lines until all saved lines have been displayed. After that, each additional CTRL/R displays the last line again.
CTRL/U	Discards the current line and the entire contents of the type-ahead buffer.
CTRL/X	Discards the current input line. It also displays the "#" character at the terminal, followed by a carriage return/line feed.

DQ$SWITCH$BUFFER

DQ$SWITCH$BUFFER substitutes a new command line for the existing one.

```
    char$offset = DQ$SWITCH$BUFFER (buff$ptr, except$ptr);
```

INPUT PARAMETER

buff$ptr	A POINTER to a STRING containing the "new" command line, that is, the one whose arguments are to be returned by subsequent calls to DQ$GET$ARGUMENT.

OUTPUT PARAMETERS

char$offset	A WORD into which the UDI places a number. This number represents the number of bytes from the beginning of the "old" command line to the last character of the last argument so far processed by DQ$GET$ARGUMENT. In other words, the value in char$offset tells how many characters in the old command line have been processed by the time of this call.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

When your program is invoked from the console, the Operating System places the invocation command into a buffer. Typically, your program will use DQ$GET$ARGUMENT to obtain the arguments in that command. If your program subsequently calls DQ$READ to obtain an additional command line from the console, it can call DQ$SWITCH$BUFFER to designate the buffer with the new command line as that from which arguments are to be obtained when DQ$GET$ARGUMENT is called.

You can use DQ$SWITCH$BUFFER any number of times to point to different strings in your program. However, you cannot use DQ$SWITCH$BUFFER to return to the command line that invoked the program, because only the Operating System knows the location of that buffer. Therefore, you should use DQ$GET$ARGUMENT to obtain all arguments of the invocation command line before issuing the first call to DQ$SWITCH$BUFFER.

A second service of DQ$SWITCH$BUFFER is that it returns the location of the last byte of the last argument so far obtained from the old buffer by calls to DQ$GET$ARGUMENT. Therefore, in addition to using DQ$SWITCH$BUFFER to switch buffers, you can use it after one or more DQ$GET$ARGUMENT calls to determine where in the buffer the next argument starts. However, doing this "resets" the buffer, in the sense that the next call to DQ$GET$ARGUMENT would return the first argument in the buffer. To return to the desired point in the buffer, where you can continue to extract arguments, call DQ$SWITCH$BUFFER again, but when doing so, use the sum of the starting address of the buffer and the value returned by the previous call to DQ$SWITCH$BUFFER. The following is an example showing how to use the second service of DQ$SWITCH$BUFFER:

```
DECLARE
    mybuffer$ptr        POINTER,
    buff$ptr            POINTER,
    arg$ptr             POINTER,
    buff                STRUCTURE(
                            offset         WORD,
                            segment        WORD) AT (@buff$ptr),
    next$char           WORD,
    char$offset         WORD,
    condition$code      WORD,
    delimit$char        BYTE;
          .
          .
          .

/* initialize buff$ptr and next$char */
    buff$ptr = mybuff$ptr;
    next$char = 0;
          .
          .
          .

/* determine where in the buffer the next argument starts */
    char$offset = DQ$SWITCH$BUFFER( buff$ptr, @condition$code );
         if condition$code <> E$OK then /* do error processing */
    next$char = char$offset + next$char;

/* return to desired point in buffer */
    buff.offset = buff.offset + char$offset;
    char$offset = DQ$SWITCH$BUFFER( buff$ptr, @condition$code );
         if condition$code <> E$OK then /* do error processing */

/* get next argument */
    delimit$char = DQ$GET$ARGUMENT( arg$ptr, @condition$ptr );
         if condition$code <> E$OK then /* do error processing */
          .
          .
          .
```

DQ$TRAP$CC

The DQ$TRAP$CC lets you specify a procedure that is to get control if an operator enters CTRL/C at the console.

```
    CALL DQ$TRAP$CC (entry$pnt, except$ptr);
```

INPUT PARAMETER

entry$pnt	A POINTER to the entry point of your CTRL/C procedure.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Normally, when an operator enters CTRL/C at the console, the system empties the type-ahead buffer and aborts the currently-executing program. By calling DQ$TRAP$CC, your program can designate any other procedure, so that it will automatically get control instead whenever CTRL/C is entered at the console.

DQ$TRAP$EXCEPTION

DQ$TRAP$EXCEPTION substitutes an alternate exception handler for the default exception handler provided by the operating system.

```
    CALL DQ$TRAP$EXCEPTION (address$ptr, except$ptr);
```

INPUT PARAMETER

address$ptr	A POINTER to a POINTER containing the entry point of the alternate exception handler.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Normally, the exception handler terminates the program that made the call producing the exception condition and displays a message to that effect on the console screen. DQ$TRAP$EXCEPTION designates an alternative exception handler as the one to which control should pass when an exceptional condition occurs.

See the section EXCEPTION-HANDLING SYSTEM CALLS at the beginning of this chapter for an explanation of the conditions of the stack when your exception handler receives control.

DQ$TRUNCATE

DQ$TRUNCATE moves the end-of-file to the current position of a named file connection's file pointer, thereby freeing the portion of the file lying beyond the file pointer.

```
    CALL DQ$TRUNCATE (connection, except$ptr);
```

INPUT PARAMETER

connection	A TOKEN for a connection to the named data file that is to be truncated. The file pointer of this connection marks the place where truncation is to occur. The byte indicated by the pointer is the first byte to be dropped from the file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call truncates a file at the current setting of the file pointer and releases all file space beyond the pointer for reallocation to other files. If the pointer is at or beyond the end of file, no truncation is performed. Unless the file pointer is already at the proper location, your program should use the DQ$SEEK system call to position the pointer before calling DQ$TRUNCATE.

The connection should have write, or read and write access rights, established when the connection was opened.

DQ$WRITE

The DQ$WRITE system call copies a collection of bytes from a buffer into a file.

```
    CALL DQ$WRITE (connection, buff$ptr, count, except$ptr;
```

INPUT PARAMETERS

connection	A WORD containing a token for the connection to the file into which the information is to be written.
buff$ptr	A POINTER to a buffer containing the data to be written to the specified file.
count	A WORD containing the number of bytes to be written from the buffer to the file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call causes the Operating System to write the specified number of bytes from the buffer to the file.

Connection Requirements

If the connection is not open for writing or updating, DQ$WRITE returns an exception code.

Number of Bytes Written

Occasionally, DQ$WRITE writes fewer bytes than requested by the calling program. This happens under the following two circumstances:

- When DQ$WRITE encounters an I/O error.
- When the volume to which your program is writing becomes full.

Where the Bytes Are Written

DQ$WRITE starts writing at the location specified by the connection's file pointer. After the writing operation is completed, the file pointer points to the byte immediately following the last byte written.

If your program must reposition the file pointer before writing, it can do so by using the DQ$SEEK system call.

***

CHAPTER 3
UDI EXAMPLE

This chapter presents an example of UDI system calls. After the program listing are the compiler and linker commands used to build the program, and a listing of the link map.

THE 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;

/* Literal declaration of TOKEN as SELECTOR */

$include(:include:ltksel.lit)

/* External declaration files for UDI system calls */

$include(:include:uexit.ext)
$include(:include:uclose.ext)
$include(:include:uwrite.ext)
$include(:include:uread.ext)
$include(:include:uopen.ext)
$include(:include:ucreat.ext)
$include(:include:ugtarg.ext)
$include(:include:uatach.ext)
$include(:include:udcex.ext)
```

```
DECLARE
    CR       LITERALLY '0DH',
    LF       LITERALLY '0AH',
    E$OK     LITERALLY '0'
    TOKEN    LITERALLY 'SELECTOR';

DECLARE
    co$conn     TOKEN;

$subtitle('check$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),
        dummy       WORD;

    IF exception <> E$OK THEN
        DO;
            CALL dq$decode$exception(exception, @exc$buf, @dummy);

            CALL dq$write(co$conn, @exc$buf.char, exc$buf.count, @dummy);

            CALL dq$write(co$conn, @(': '), 2, @dummy);

            CALL dq$write(co$conn, @info.char, info.count, @dummy);

            CALL dq$write(co$conn, @(CR, LF), 2, @dummy);

            CALL dq$exit(3);
        END;

    END check$exception;
```

```
$subtitle('Main')
/*.....................................................................
 *
 *         --- MAIN PROGRAM ---
 *
 *.....................................................................
 */


    DECLARE st WORD;

    DECLARE
        in$name(50)     BYTE,
        out$name(50)    BYTE,
        in$conn         TOKEN,
        out$conn        TOKEN,
        delim           BYTE;

    DECLARE
        buffer(1024)    BYTE,
        in$bp           POINTER,
        in$char         BASED in$bp BYTE,
        nextchar        BASED in$bp (2) BYTE,
        in$count        WORD,
        i               WORD;


    /*.................................................................
     *  Create a connection to :CO: (console output).
     *.................................................................
     */

    co$conn = dq$create(@(4, ':CO:'), @st);

    CALL dq$open(co$conn, 2, 0, @st);


    /*.................................................................
     *  Ignore the name of the program (the first argument).
     *.................................................................
     */

    delim = dq$get$argument(@buffer, @st);
    CALL check$exception(st, 0);
    IF delim = CR THEN
        CALL dq$exit(0);
```

```
/*.......................................................................
 *  Attach the input file, and open it.
 *.......................................................................
 */

delim = dq$get$argument(@in$name, @st);
CALL check$exception(st, 0);

in$conn = dq$attach(@in$name, @st);
CALL check$exception(st, @in$name);

CALL dq$open(in$conn, 1, 2, @st);
CALL check$exception(st, @in$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, @st);
        CALL check$exception(st, 0);
        IF (delim = CR) OR
           (buffer(0) <> 2) OR
           (buffer(1) <> 'T') OR
           (buffer(2) <> 'O') THEN
            DO;
                CALL dq$write(co$conn, @('Invalid output file', CR,
                    LF), 21, @st);
                CALL dq$exit(3);
            END;

        delim = dq$get$argument(@out$name, @st);
        CALL check$exception(st, 0);

        out$conn = dq$create(@out$name, @st);
        CALL check$exception(st, @out$name);

        CALL dq$open(out$conn, 2, 2, @st);
        CALL check$exception(st, @out$name);
    END;
ELSE
    out$conn = co$conn;
    /*.................................................................
     *  Read from input, convert, and write to output
     *.................................................................
     */

    DO WHILE 1;
        in$count = dq$read(in$conn, @buffer, size(buffer), @st);
        CALL check$exception(st, @in$name);
        IF in$count = 0 THEN
            GOTO end$of$file;

        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$count, @st);
        CALL check$exception(st, @out$name);
    END;
end$of$file:

    /*.................................................................
     *  Close input and output files, and exit
     *.................................................................
     */

    CALL dq$close(in$conn, @st);
    CALL check$exception(st, @in$name);

    CALL dq$close(out$conn, @st);
    CALL check$exception(st, @out$name);

    CALL dq$exit(0);

END upper;
```

COMPILING AND LINKING

The program UPPER was compiled and linked on an iRMX 86-based system with the following commands:

```
    attachfile :sd:lib/rmx86 as :lib:
    plm86 upper.p86
    link86 upper.obj, :lib:compac.lib  to upper  bind mempool(5000H)
```

The link map is on the next page.

```
iRMX 86 8086 LINKER, V2.0

INPUT FILES: UPPER.OBJ, :LIB:COMPAC.LIB
OUTPUT FILE: UPPER
CONTROLS SPECIFIED IN INVOCATION COMMAND:
  BIND MEMPOOL(5000H)
DATE:  14/02/83  TIME:  12:05:37

LINK MAP OF MODULE UPPER

LOGICAL SEGMENTS INCLUDED:
LENGTH ADDRESS ALIGN  SEGMENT         CLASS            OVERLAY
 02F6H ------    W    CODE            CODE
 001EH ------    W    CONST           CONST
 0475H ------    W    DATA            DATA
 0454H ------    W    STACK           STACK
 0000H ------    W    MEMORY          MEMORY
 0000H ------    G    ??SEG

INPUT MODULES INCLUDED:
UPPER.OBJ(UPPER)
:LIB:COMPAC.LIB(DQATTACH)
:LIB:COMPAC.LIB(DQCLOSE)
:LIB:COMPAC.LIB(DQCREATE)
:LIB:COMPAC.LIB(DQDECODEEXCEPTION)
:LIB:COMPAC.LIB(DQEXIT)
:LIB:COMPAC.LIB(DQGETARGUMENT)
:LIB:COMPAC.LIB(DQOPEN)
:LIB:COMPAC.LIB(DQREAD)
:LIB:COMPAC.LIB(DQWRITE)
:LIB:COMPAC.LIB(SYSTEMSTACK)



GROUP MAP

GROUP NAME:  CGROUP
   OFFSET  SEGMENT NAME
   0000H   CODE

GROUP NAME:  DGROUP
   OFFSET  SEGMENT NAME
   0000H   CONST
   001EH   DATA

SYMBOL TABLE OF MODULE UPPER

BASE    OFFSET TYPE SYMBOL               BASE    OFFSET TYPE SYMBOL

G(1)    0293H  PUB  DQATTACH             G(1)    029EH  PUB  DQCLOSE
G(1)    02A9H  PUB  DQCREATE             G(1)    02B4H  PUB  DQDECODEEXCEPTION
G(1)    02BFH  PUB  DQEXIT               G(1)    02CAH  PUB  DQGETARGUMENT
G(1)    02D5H  PUB  DQOPEN               G(1)    02E0H  PUB  DQREAD
G(1)    02EBH  PUB  DQWRITE              S(4)    006CH  PUB  SYSTEMSTACK
```

***

APPENDIX A DATA TYPES

The following data types are recognized by the iRMX 86 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 address of a (64K-byte processor) segment and an offset in the segment. The offset is in the word having the lower address.
OFFSET	A word whose value represents the distance from the base address of a segment.
SELECTOR	The base address of a segment.
TOKEN	A word or selector whose value identifies an object. A token can be declared literally a WORD or a SELECTOR depending on your needs.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes that follow it in the string.
DWORD	A 4-byte unsigned binary number.

***

APPENDIX B
iRMX™ 86 CONDITION CODES

This appendix contains the exception codes that are generated by the iRMX 86 Operating System. Exception codes are any condition codes other than E$OK, the normal code. Exception codes are classed as either "Environmental Conditions" or "Programmer Errors", although the latter includes certain hardware errors as well as errors that result from programming.

The values of these exception codes fall into ranges based on the iRMX 86 layer which first detects the condition. Table B-1 lists the layers and their respective ranges, with numeric values expressed in hexadecimal notation.

Table B-1. Exception 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 iRMX 86 NUCLEUS REFERENCE MANUAL gives the value of each code and its associated mnemonic, as well as a short description of its significance. In addition, the table shows the layer(s) of the system that could generate the code, in case you wish to refer the the appropriate manual.

***

INDEX

Primary references are underscored.

***

GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX™ 86 AND iRMX™ 88 I/O SYSTEMS

CONTENTS

CONTENTS (continued)

***

CHAPTER 1 INTRODUCTION

The iRMX 86 and iRMX 88 I/O Systems are each 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, an iSBC 215 device driver provides the facilities that enable you to use an iSBC 215 Generic Winchester controller to control a Winchester-type 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 Systems are 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. Figure 1-1 shows these levels of communication.

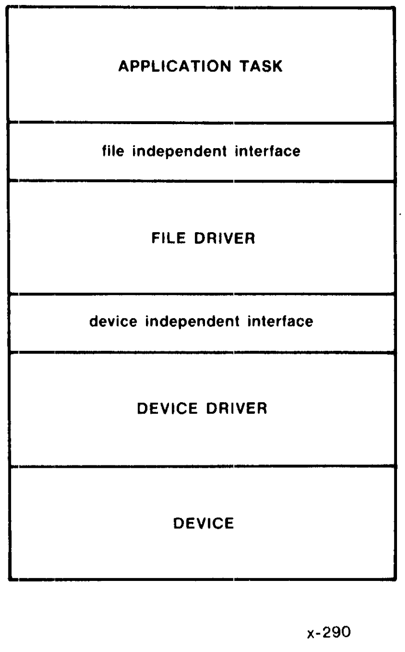


Figure 1-1. Communication Levels

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.

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 <u>device number</u> identifies the controller among all the controllers in the system, the <u>unit number</u> identifies the unit within the device, and the unique <u>device-unit number</u> 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.

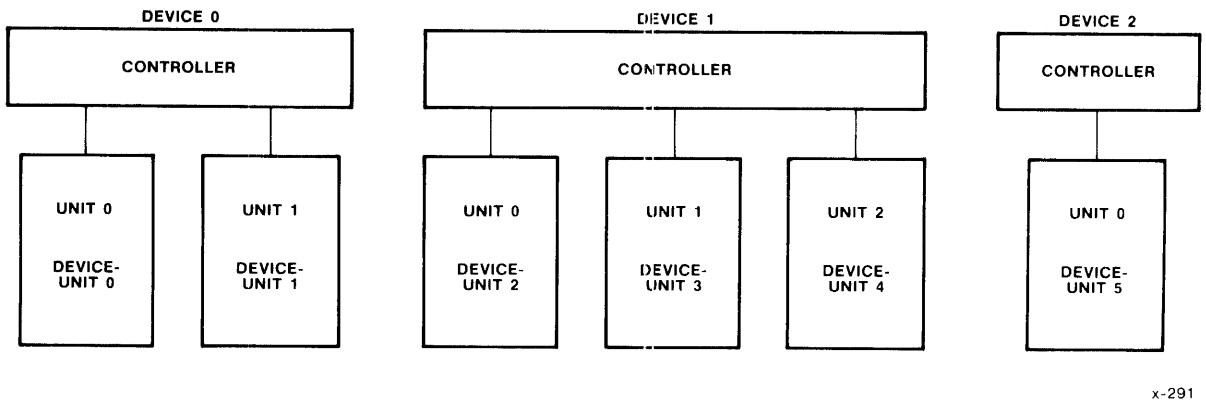


Figure 1-2. Device Numbering

You must provide a device driver for every device in your 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 215 controllers are two separate devices and each has its own device driver. However, these device drivers can share common code.)

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 operation. Operations supported by the I/O System are:

Read
Write
Seek
Special
Attach device
Detach device
Open
Close

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 Chapter 2.) The device driver must translate this request into specific device commands to cause the device to perform the requested operation.

TYPES OF DEVICE DRIVERS

The I/O System supports four types of device drivers: custom, common, random access, and terminal. A custom device driver is one that the user creates in its entirety. This type of device driver can assume any form and can provide any functions that the user wishes, 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, and Cancel I/O.

The I/O System provides the basic support routines for the common, random access, and terminal device driver types. These support routines provide a queueing mechanism, an interrupt handler, and other features needed by common, random access, and terminal devices. If your device fits into the common, random access, or terminal device classification, you need to write only the specialized, device-dependent procedures and interface them to the ones provided by the I/O System to create a complete device driver.

HOW TO READ THIS MANUAL

This manual is for people who plan to write device drivers for use with iRMX 86- and/or iRMX 88-based systems. Because there are numerous terminology differences between the two iRMX systems, the tone of this manual is general, unlike that of other manuals for either system. For iRMX 88 users, this should not be a problem. But iRMX 86 users should take note of the following:

- In a number of places the phrase "the location of" is substituted for "a token for".
- The "device data storage area" that is alluded to in many places is actually an iRMX 86 segment.
- The term "resources" usually means "objects." The intended meaning of "resources" is clear from its context.

***

CHAPTER 2
DEVICE DRIVER INTERFACES

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 which 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 these interfaces.

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).

DEVICE-UNIT INFORMATION BLOCK (DUIB)

The DUIB is an interface between a device driver and the I/O System, in the sense that the DUIB contains the addresses of one of the following 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/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.

DUIB Structure

This section lists the elements that make up a DUIB. When creating DUIBs for iRMX 86 applications, code them in the format shown here (as assembly-language structures). The iRMX 86 Interactive Configuration Utility (ICU) includes your DUIB file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the structure.

Unlike the iRMX 86 ICU, the iRMX 88 ICU prompts you for some fields in the DUIB structure. The ICU automatically fills in the other fields, depending upon factors such as the type of device you are configuring. The iRMX 88 ICU generates the DUIBs and places them in the device configuration source file.

```
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$UPDATE,                    ; byte (iRMX 86 DUIB only)
&    MAX$BUFFERS,                     ; byte (iRMX 86 DUIB only)
&    RESERVED,                        ; byte (iRMX 86 DUIB only)
&    >
```

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.

You supply the name when configuring your application system. If you are an iRMX 86 user, you specify the DUIB name when attaching a unit via the RQ$A$PHYSICAL$ATTACH$DEVICE system call. Device drivers can ignore this field.

For the iRMX 88 Executive, the DUIB name is the device name portion of the name$p parameter for the DQ$ATTACH or the DQ$CREATE system calls.

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 (iRMX 86 only)
3	named

The remaining bits of the word must be set to zero. Device drivers can ignore 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 can ignore 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 not significant 1 = bits 1-7 significant
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, meaning that track 0 is single-density with 128-byte sectors 1 = not a standard diskette or not a diskette
5-7	reserved

If bit 0 is set to 1, then a driver for the device can read track 0 when asked to do so by the I/O System.

DEV$GRAN	WORD specifying the device granularity, in bytes. This parameter applies to random access devices. It specifies the minimum number of bytes of information that the device reads or writes in one operation.If the device is a disk or magnetic bubble device, you should set this field equal to the sector size for the device. Otherwise, set this field equal to zero.
DEV$SIZE	DWORD specifying the number of bytes of information that the device-unit can store.
DEVICE	BYTE specifying the device number of the device with which this device-unit is associated. Device drivers can ignore 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 address of the Initialize I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
FINISH$IO	WORD specifying the address of the Finish I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
QUEUE$IO	WORD specifying the address of the Queue I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
CANCEL$IO	WORD specifying the address of the Cancel I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
DEVICE$INFO$P	POINTER to a structure which contains additional information about the device. The common, random access, and terminal device drivers require, for each device, a Device Information Table, in a particular format. This structure is described in Chapter 3. If you are writing a custom driver, you can place information in this structure depending on the needs of your driver. Specify a zero for this parameter if the associated device driver does not use this field.
UNIT$INFO$P	POINTER to a structure that contains additional information about the unit. Random access and terminal device drivers require this Unit Information Table in a particular format. Refer to Chapter 3 for further information. 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$TIMEOUT	WORD specifying the number of system time units that the I/O System must wait before writing a partial sector after processing a write request for a disk device. In the case of drivers for devices that are neither disk nor magnetic bubble devices, set this field to 0FFFFH during configuration. This field applies only to the device for which this is a DUIB, and is independent of updating that is done either because of the value in the FIXED$UPDATE field of the DUIB or by means of the A$UPDATE system call of the I/O System. Device drivers can ignore this field.
NUM$BUFFERS	WORD which, if not zero, both specifies that the device is a random access device and indicates the number of buffers 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 that the device is not a random access device. Device drivers can ignore this field.
PRIORITY	BYTE specifying the priority of the I/O System service task for the device. Device drivers can ignore this field.
FIXED$UPDATE	BYTE indicating whether the fixed update option was selected for the device when the application system was configured. This option, when selected, causes the I/O System to finish any write requests that had not been finished earlier because less than a full sector remained to be written. Fixed updates are performed throughout the entire system whenever a time interval (specified during configuration) elapses. This is independent of the updating that is indicated for a particular device (by the UPDATE$TIMEOUT field of the DUIB) or the updating of a particular device that is indicated by the A$UPDATE system call of the I/O System. A value of 0FFH indicates that fixed updating has been selected for this device, and a value of zero indicates that it has not been selected. Device drivers can ignore this field. The FIXED$UPDATE field is not present in the iRMX 88 DUIB.

MAX$BUFFERS	BYTE specifying the maximum number of buffers that the Extended I/O System (of the iRMX 86 Operating System) can allocate for a connection to this device when the connection is opened by a call to S$OPEN. The value in this field is specified during configuration. Device drivers can ignore this field. The MAX$BUFFERS field is not present in the iRMX 88 DUIB.
RESERVED	BYTE reserved for future use. The RESERVED field is not present in the iRMX 88 DUIB.

Using the DUIBs

To use the I/O System to communicate with files on a device-unit, you must first attach the unit. If you are an iRMX 88 user, attaching the unit occurs automatically when you first attach or create a file on the unit. If you are an iRMX 86 user, you attach the unit by invoking the RQ$A$PHYSICAL$ATTACH$DEVICE system call (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for a description of this system call).

When you attach a unit, the I/O System assumes that 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/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$DEVICE system call (for iRMX 86 users) or the appropriate device name when calling DQ$ATTACH or DQ$CREATE (for iRMX 88 users.) However, before you can switch the DUIBs for a unit, you must detach the unit.

Figure 2-1 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 2-1, 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

For iRMX 86 systems only, 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 86 applications, you must supply a separate DUIB for every kind of volume you intend to format via the FORMAT Human Interface command.

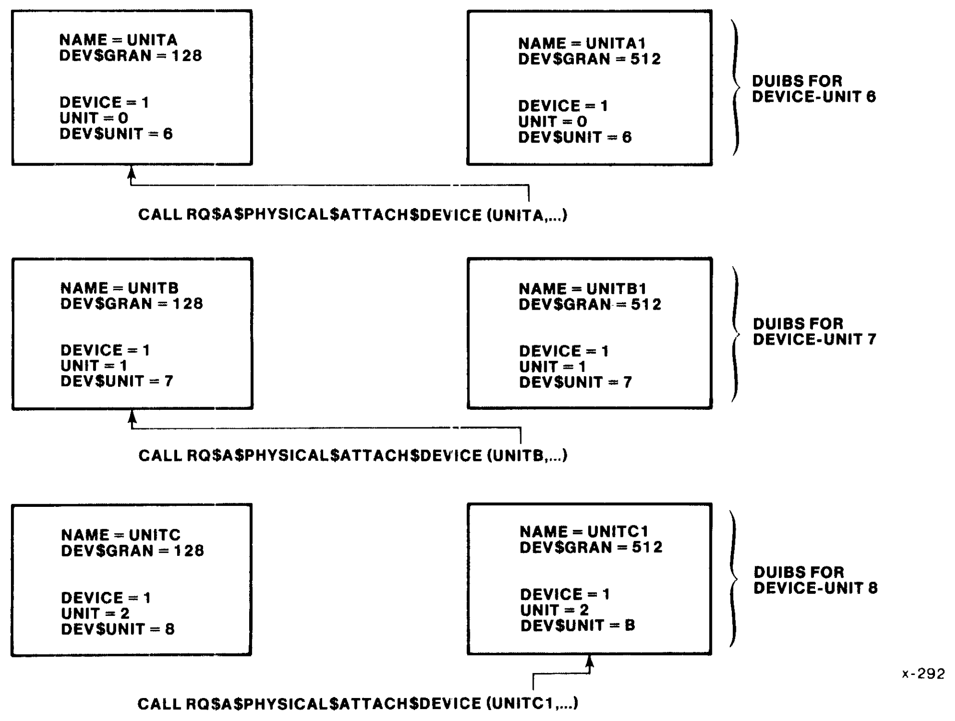


Figure 2-1. Attaching Devices

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/device driver support 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/device driver support procedures in all of the DUIBs associated with a particular device. There is only one set of device driver/device driver support 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 4 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.
- For iRMX 86 systems only, if you write a terminal driver, you must supply terminal device information table for each terminal device driver, as well as a unit information table for each terminal. See Chapter 7 for specifications of these tables.

I/O REQUEST/RESULT SEGMENT (IORS)

An I/O request/result segment (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 a user 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 the appropriate device driver, which then processes the request. When the device driver performs the operation indicated in the IORS, it must modify the IORS to indicate what it has done and send the IORS back to the response mailbox (exchange) indicated in the IORS.

The IORS is the only mechanism that the I/O System uses to transmit requests to device drivers. The IORS structure is always the same. Every device driver must be aware of this structure and must update the information in the IORS after performing the requested function. 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$MBOX       SELECTOR,
          DONE            BYTE,
          FILL            BYTE,
          CANCEL$ID       SELECTOR,
          CONN$T          SELECTOR); (iRMX 86 IORS only)
```

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 iRMX 86 NUCLEUS REFERENCE MANUAL, the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL, and the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL, or the iRMX 88 REFERENCE 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
4	IO$WRPROT	Write-protected volume
5*	IO$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

*For iRMX 86 systems only.

The I/O System reserves values 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 that you wish. For example, the iSBC 204 driver places the controller's result byte in the high eight bits of this field. For more information about the data returned by your device controller, refer to the hardware reference manual for your controller.

ACTUAL — WORD which 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 upon the file driver to be 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

File Driver For Connection	Subfunct Value	Function
Physical	4	Get terminal data
Physical	5	Set terminal data
Physical	6	Set signal
Physical	7	Rewind tape
Physical	8	Read tape file mark
Physical	9	Write tape file mark
Physical	10	Retension tape
	11-32767	Reserved for other Intel products

*These functions apply both to iRMX 86 and iRMX 88 systems. The other functions are iRMX 86-specific.

The values from 32768 to 65535 are available for user-written/custom device drivers.

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 support routines.

If the device driver is a random access driver, the random access support routines modify the information in the DEV$LOC field before passing the IORS on to user-written driver procedures listed in Chapter 5. The value that the random access support routines fill out depends upon the TRACK$SIZE field in the unit's Unit Information Table (see Chapter 3).

- 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$SIZE field is nonzero, the random access support routines use the absolute byte number in DEV$LOC 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 which the I/O System sets to indicate the internal buffer where data is read from or written to.

COUNT	WORD which the I/O System sets to indicate the number of bytes to transfer.
COUNT$FILL	Reserved WORD.
AUX$P	POINTER which 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 that 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 special, SUBFUNCT equals format track, 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$number	The number of the track to be formatted. Acceptable values are 0 to (number of tracks on the volume - 1).
interleave	The interleaving factor for the track. (That is, the number of physical sectors to advance when locating the next logical sector.) The supplied value, before being used, is evaluated MOD the number of sectors per track.
track$offset	The number of physical sectors to advance when locating the first logical sector on the next track.
fill$char	The byte value with which each sector is to be filled.

NOTE

The rest of the information about the AUX$P field is iRMX 86-specific.

In a request to set up an iRMX 86 mailbox, where the iRMX 86 I/O System is to send 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 special, SUBFUNCT equals notify, and AUX$P points to a structure of the form:

```
DECLARE SETUP$NOTIFY STRUCTURE(
            MAILBOX         SELECTOR,
            OBJECT          SELECTOR);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL. Random access drivers do not have to process such requests because they are handled by the I/O System.

In a request to obtain information about iSBC 215 or iSBC 220 (supported) disk devices, FUNCT equals special, SUBFUNCT equals get device characteristics, 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 iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

In a request to obtain information about iSBX 217 tape drives (associated with an iSBC 215G board), FUNCT equals special, SUBFUNCT equals get device characteristics, and AUX$P points to a structure of the form:

```
DECLARE TAPE$DRIVE$DATA STRUCTURE(
     TAPE                   WORD,
     RESERVED(7)            BYTE);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

In a request to read or write terminal mode information for a terminal being driven by a terminal driver, FUNCT equals special, SUBFUNCT equals get terminal attributes (for reading) or set terminal attributes (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,
     FLOW$CONTROL         WORD,
     HIGH$WATER$MARK      WORD,
     LOW$WATER$MARK       WORD,
     FC$ON$CHAR           WORD,
     FC$OFF$CHAR          WORD);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL. If you are using the Terminal Support Code, this special subfunction is invisible to the terminal device driver.

In a request to set up special character recognition in the input stream of a terminal driver for signalling purposes, FUNCT equals special, SUBFUNCT equals signal, and AUX$P points to a structure of the form:

```
DECLARE SIGNAL$CHARACTER STRUCTURE(
     SEMAPHORE            SELECTOR
     CHARACTER            BYTE);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL. In a request to read a tape file mark, FUNCT equals special, SUBFUNCT equals read tape file mark, and AUX$P points to a structure of the form:

```
DECLARE READ$FILE$MARK STRUCTURE(
     SEARCH               BYTE);
```

where the field is defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

LINK$FOR	POINTER that the device driver/device driver support routines can use to implement a request queue. This field points to the location of the next IORS in the queue.
LINK$BACK	POINTER that the device driver/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$MBOX	SELECTOR that the I/O System fills with either an iRMX 86 token for the response mailbox or the address of an iRMX 88 exchange. Upon completion of the I/O request, the device driver/device driver support routines must send the IORS to this response mailbox or exchange.
DONE	BYTE that the device driver can set to TRUE (0FFH) or FALSE (00H) to indicate whether the entire request has been completed.
FILL	Reserved BYTE.
CANCEL$ID	SELECTOR used to identify queued I/O requests that CANCEL$IO can remove from the queue.
CONN$T	SELECTOR used in requests to the iRMX 86 I/O System. This field contains the token of the iRMX 86 file connection through which the request was issued.

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 that a procedure of this sort must go through vary considerably, depending on the type of I/O device. Procedures supplied with the I/O System to manipulate devices such as the iSBC 204 and iSBC 206 devices use the PL/M-86 builtins INPUT and OUTPUT to transmit to and receive from I/O ports. Other devices may require different methods. The I/O System places no restrictions on the method of communicating with devices. Use the method that the device requires.

***

CHAPTER 3 CATEGORIES AND PROPERTIES OF DEVICES AND DRIVERS

There are four types of device drivers in the iRMX 86 environment: common, random access, custom, and terminal. There are three types of device drivers in the iRMX 88 environment: common, random access, and custom. This chapter defines the distinctions between the types of drivers and discusses the characteristics and data structures pertaining to common and random access device drivers. Chapters 5, 6, and 7 are devoted to explaining how to write the various types of device drivers.

CATEGORIES OF DEVICES

Because the I/O System provides procedures that constitute the bulk of any common or random access device driver, you should consider the possibility that your device is a common or random access device. If your device falls in either of these categories, you can avoid most of the work of writing a device driver by using the supplied procedures. The following sections define the four types of devices.

COMMON DEVICES

Common devices are relatively simple devices other than terminals, such as line printers. This category includes devices that 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 a device.
- Data either read or written by these devices does not need to be broken up into blocks.

If you have a device that fits into this category, you can save the effort of creating an entire device driver by using the common driver routines supplied by the I/O System. Chapter 5 of this manual describes the procedures that you must write to complete the balance of a common device driver.

RANDOM ACCESS DEVICES

A random access device is a device, such as a disk drive, in which data can be read from or written to any address of the device. The support routines provided by the I/O System for random access assume 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.
- I/O requests must be broken up into blocks of a specific length.
- The device supports random access seek operations.

If you have devices that fit into the random access category, you can take advantage of the random access support routines provided by the I/O System. Chapter 5 of this manual describes the procedures that you must write to complete the balance of a random access device driver.

TERMINAL DEVICES

A terminal device is characterized by the fact that it reads and writes single characters, with an interrupt for each character. Because such devices are entirely different than common, random access, and even custom devices, terminal drivers and their required data structures are described in Chapter 7. The remainder of this chapter applies only to common, random access, and custom device drivers.

CUSTOM DEVICES

If your device fits neither the common nor the random access category, and is not a terminal or terminal-like device, you must write the entire driver for the device. The requirements of a custom device driver are defined in Chapter 6.

I/O SYSTEM-SUPPLIED ROUTINES FOR COMMON AND RANDOM ACCESS DEVICE DRIVERS

The I/O System supplies the common and random access routines that it calls when processing I/O requests. Flow charts for these procedures appear in Appendix A. The names and functions of these procedures are as follows: (The "RAD$" prefix applies to iRMX 88 routine names.)

Routine	Function
(RAD$)INIT$IO	Creates the resources needed by the remainder of the driver routines, creates an interrupt task, and calls a user-supplied routine that initializes the device itself.
(RAD$)FINISH$IO	Deletes the resources used by the other driver routines, deletes the interrupt task, and calls a user-supplied procedure that performs final processing on the device itself.

Routine	Function
(RAD$)QUEUE$IO	Places I/O requests (IORSs) on the queue of requests.
(RAD$)CANCEL$IO	Removes one or more requests from the request queue, possibly stopping the processing of a request that has already been started.

These routines process I/O requests for both common and random access devices. They distinguish between categories 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.

In addition to the routines just described, the I/O System supplies an interrupt handler (interrupt service routine) and an interrupt task (called INTERRUPT$TASK) which respond to all interrupts generated by the units of a device, process those interrupts, and start the device working on the next I/O request on the queue. The INIT$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 either processes the interrupt itself or invokes an interrupt task to process the interrupt. Since an interrupt handler is limited in the types of system calls that it can make and the number of interrupts that can be enabled while it is processing, an interrupt task usually services the interrupt. 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 3-1 shows the interaction between an interrupt task, an I/O device, an I/O request queue, and the Queue I/O 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 I/O procedure runs in parallel, putting additional I/O requests on the queue.

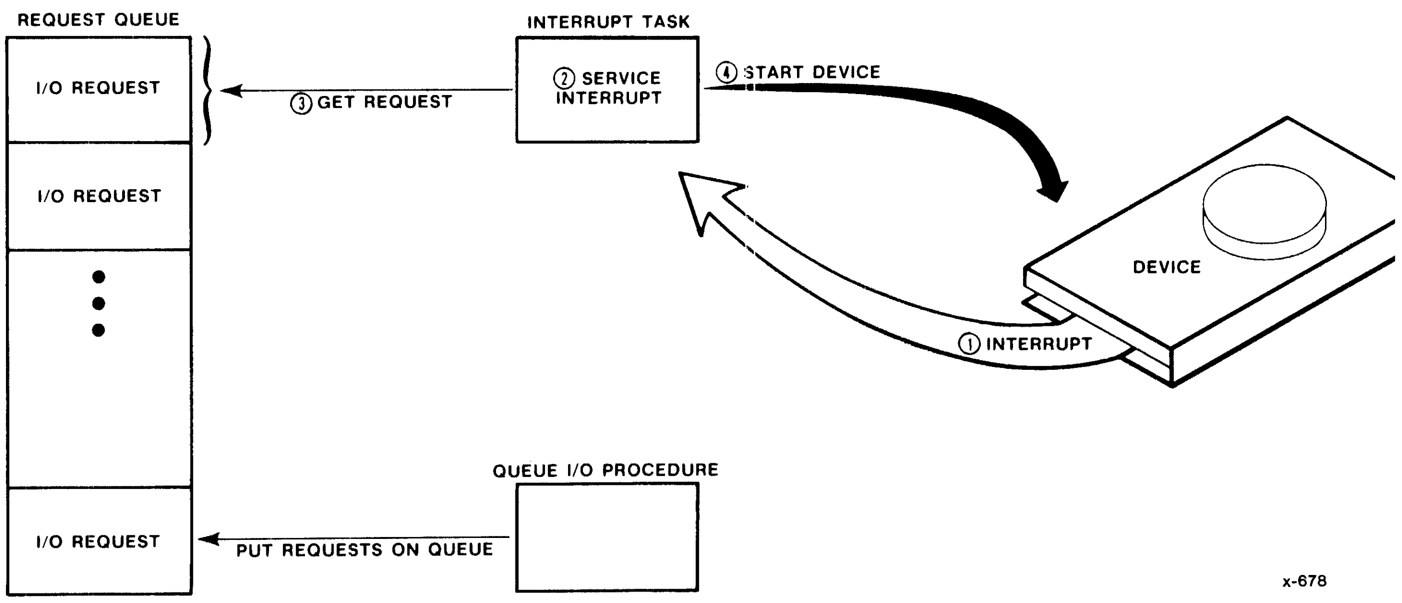


Figure 3-1. Interrupt Task Interaction

I/O SYSTEM ALGORITHM FOR CALLING THE DEVICE DRIVER PROCEDURES

The I/O System calls each of the four device driver procedures in response to specific conditions. Figure 3-2 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 3-2 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, if you are an iRMX 86 user, the first I/O request to each device-unit must be an RQ$A$PHYSICAL$ATTACH$DEVICE system call, and if you are an iRMX 88 user, the first request to each device-unit must be either a DQ$ATTACH or a DQ$CREATE system call.

2. If the request results from an RQ$A$PHYSICAL$ATTACH$DEVICE, a DQ$ATTACH, or a DQ$CREATE 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 Initialize I/O procedure first, before queueing the request.

3. Whether or not the I/O System called the Initialize I/O procedure, it calls the Queue I/O procedure to queue the request for execution.

4. If you are an iRMX 86 user and the request just queued resulted from an iRMX 86 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 I/O procedure to do any final processing on the device and clean up resources used by the device driver routines.

   If you are an iRMX 88 user and the request just queued resulted from either a DQ$DETACH or a DQ$DELETE 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 I/O procedure to do any final processing on the device and clean up resources used by the device driver routines.

The iRMX 86 I/O System calls the fourth device driver procedure, the Cancel I/O 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 iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL describes the hard detach option.

- If the job containing the task which made a request is deleted.

The iRMX 88 I/O System does not call the Cancel I/O procedure.

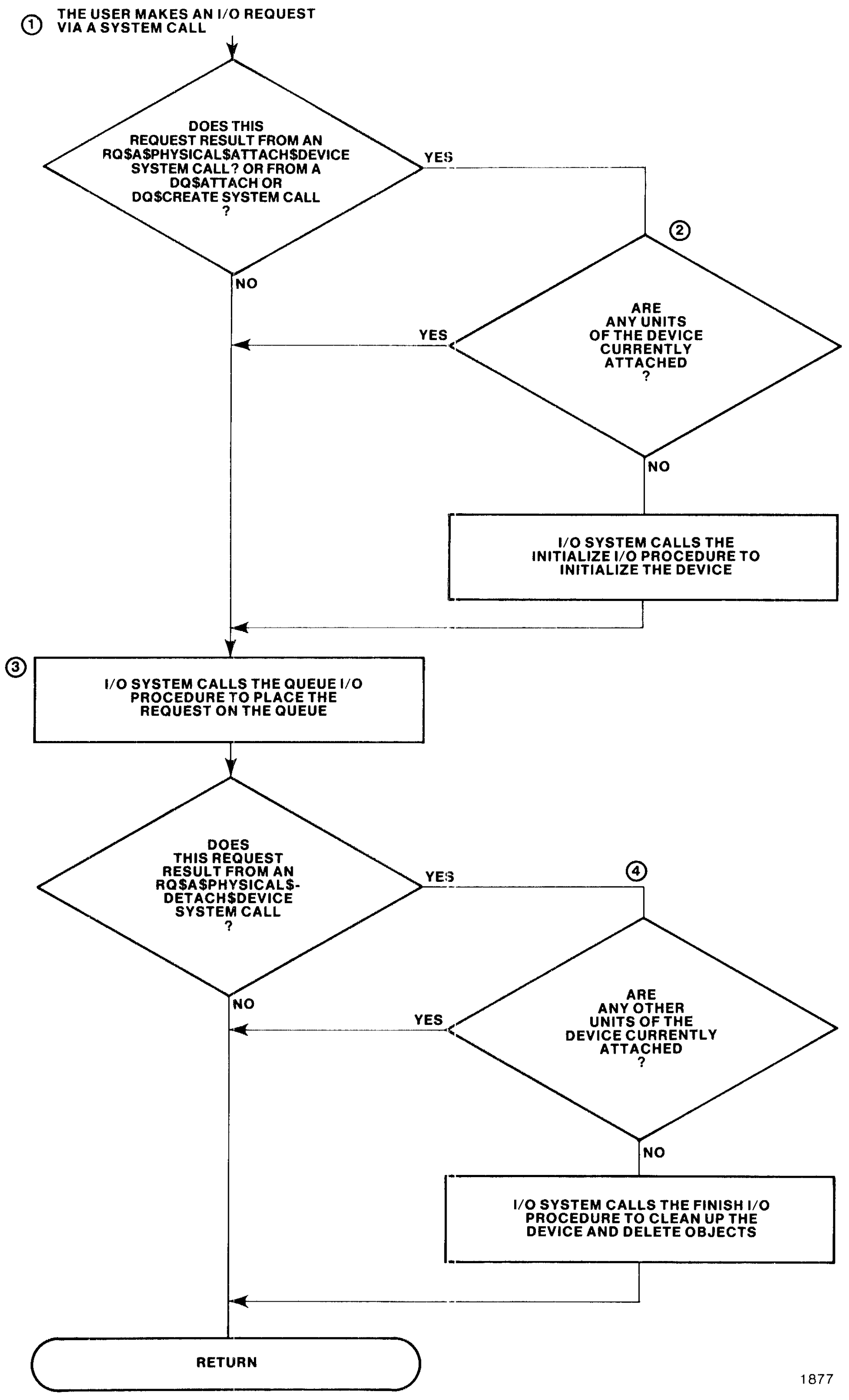

① THE USER MAKES AN I/O REQUEST VIA A SYSTEM CALL
DOES THIS REQUEST RESULT FROM AN RQ$A$PHYSICAL$ATTACH$DEVICE SYSTEM CALL? OR FROM A DQ$ATTACH OR DQ$CREATE SYSTEM CALL ?
YES
NO
②
ARE ANY UNITS OF THE DEVICE CURRENTLY ATTACHED ?
YES
NO
I/O SYSTEM CALLS THE INITIALIZE I/O PROCEDURE TO INITIALIZE THE DEVICE
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
④
NO
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 3-2. How the I/O System Calls the Device Driver Procedures

REQUIRED DATA STRUCTURES

In order for the I/O System-supplied routines to be able to call the user-supplied routines, you must supply the addresses of these user-supplied routines, as well as other information, in a Device Information Table. In addition, processing I/O requests through a random access driver requires a Unit Information Table. Each DUIB contains one pointer field for a Device Information Table and another for a Unit Information Table.

DUIBs that correspond to units of the same device should point to the same Device Information Table, but they can point to different Unit Information Tables, if the units have different characteristics. Figure 3-3 illustrates this.

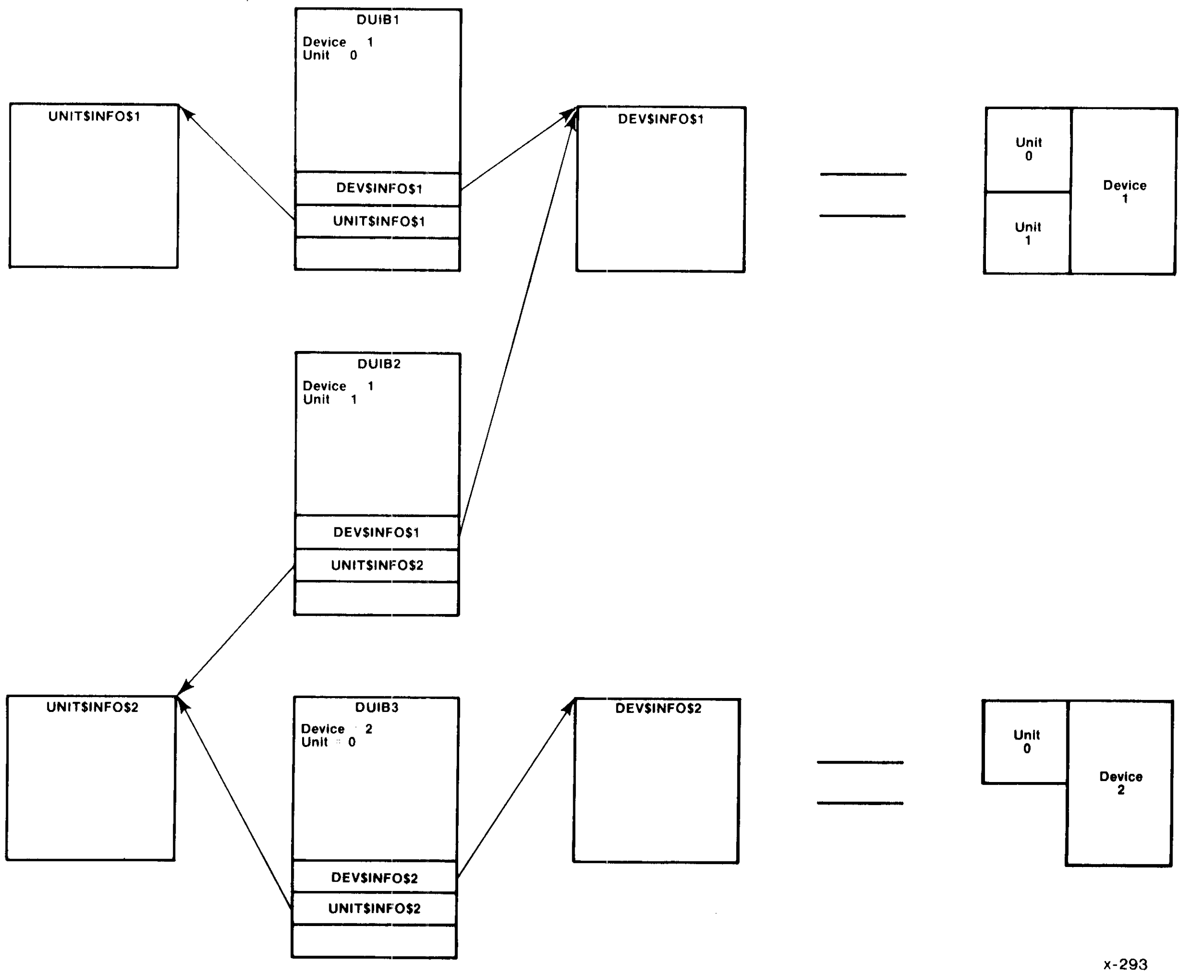


Figure 3-3. DUIBs, Device and Unit Information Tables

DEVICE INFORMATION TABLE

Common and random access Device Information Tables contain the same fields in the same order. When creating Device Information Tables for iRMX 86 applications, code them in the format shown here (as assembly-language structures). If you give the iRMX 86 ICU the pathname of your Unit Information Table file, the ICU includes the file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the structure.

The fields DEVICE$INIT, DEVICE$FINISH, DEVICE$START, DEVICE$STOP, and DEVICE$INTERRUPT contain the names of user-supplied procedures whose duties are described in Chapter 5. When creating the file containing your Device Information Tables, specify external declarations for these user-supplied procedures. This allows the code for these user-supplied procedures to be included into the assembly of the I/O System. For example, if your procedures are named DEVICE$INIT, DEVICE$FINISH, DEVICE$START, DEVICE$STOP, and DEVICE$INTERRUPT, include the following declarations in the file containing your Device Information Tables:

```
extrn device$init: near
extrn device$finish: near
extrn device$start: near
extrn device$stop: near
extrn device$interrupt: near
```

The iRMX 88 ICU prompts you for each field in the Device Information Table structure. The iRMX 88 ICU generates the Device Information Table and places it in the device configuration source file.

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
&    >
```

where:

LEVEL	WORD specifying 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:

<u>iRMX 86 VALUES</u>

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

<u>iRMX 88 VALUES</u>

The values available are 0 through 3FH. Refer to the iRMX 88 REFERENCE MANUAL for further information.

PRIORITY	BYTE specifying the initial priority of the interrupt task. The actual priority of an iRMX 86 interrupt task might change because the iRMX 86 Nucleus adjusts an interrupt task's priority according to the interrupt level that it services. Refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for further information about this relationship between interrupt task priorities and interrupt levels.
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. This then is the size of the read or write buffers plus any flags that the user-written routines need.
NUM$UNITS	WORD specifying the number of units supported by the driver. Units are assumed to be numbered consecutively, starting with zero.

DEVICE$INIT	WORD specifying the start address of a user-written device initialization procedure. The format of this procedure, which INIT$IO calls, is described in Chapter 5.
DEVICE$FINISH	WORD specifying the start address of a user-written device finish procedure. The format of this procedure, which FINISH$IO calls, is described in Chapter 5.
DEVICE$START	WORD specifying the start address of a user-written device start procedure. The format of this procedure, which QUEUE$IO and INTERRUPT$TASK call, is described in Chapter 5.
DEVICE$STOP	WORD specifying the start address of a user-written device stop procedure. The format of this procedure, which CANCEL$IO calls, is described in Chapter 5.
DEVICE$INTERRUPT	WORD specifying the start address of a user-written device interrupt procedure. The format of this procedure, which INTERRUPT$TASK calls, is described in Chapter 5.

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.

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.

When creating Unit Information Tables for iRMX 86 applications, code them in the format shown here (as assembly-language structures). If you give the iRMX 86 ICU the pathname of your Unit Information Table file, the ICU includes the file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the structure.

The iRMX 88 ICU prompts you for some fields in the Unit Information Table structure. The iRMX 88 ICU generates the Unit Information Table and places it in the device configuration source file.

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$SIZE	WORD specifying the size, in bytes, of a single track of a volume on the unit. If the device controller supports reading and writing across track boundaries, and your driver is a random-access driver, place a zero in this field. If you specify a zero for this field, the I/O System-supplied random access support procedures place an absolute sector number in the DEV$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 in Chapter 5.
MAX$RETRY	WORD specifying the maximum number of times an I/O request should be tried if an error occurs. Nine is the recommended value for this field. When this field contains a nonzero value, the I/O System-supplied procedures guarantee that read or write requests are retried if the user-supplied device start or device interrupt procedures return an IO$SOFT condition in the IORS.UNIT$STATUS field. (The IORS.UNIT$STATUS field is described in the "IORS Structure" section of Chapter 2.)
CYLINDER$SIZE	For iRMX 86 systems, a WORD whose meaning depends on its value, as follows:

0	The I/O System never requests a seek operation. Instead, it expects the device driver/controller to perform implied "seeks" when a read/write on the unit begins on a cylinder which is different from the one associated with the current position of the read/write head.
1	The I/O System automatically requests a seek operation (to seek to the correct cylinder) before performing a 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 automatically requests a seek operation whenever a requested read or write operation on the unit 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.

RELATIONSHIPS BETWEEN I/O PROCEDURES AND I/O DATA STRUCTURES

This section brings together several of the procedures and data structures that have been described so far in this manual. Figure 3-4 shows the many relationships that exist among these entities, with solid arrows indicating procedure calls and dotted arrows indicating pointers. Note that the I/O System contains the address of each DUIB, which in turn contains the addresses of the procedures that the I/O System calls when performing I/O on the associated device-unit. The DUIB also contains the address of the Device Information Table and, if the device is a random access device, the Unit Information Table. The Device Information Table, in turn, contains the addresses of the procedures that are called by the procedures that the I/O System calls. It is through these links that the appropriate calls are made in the servicing of an I/O request for a particular device-unit.

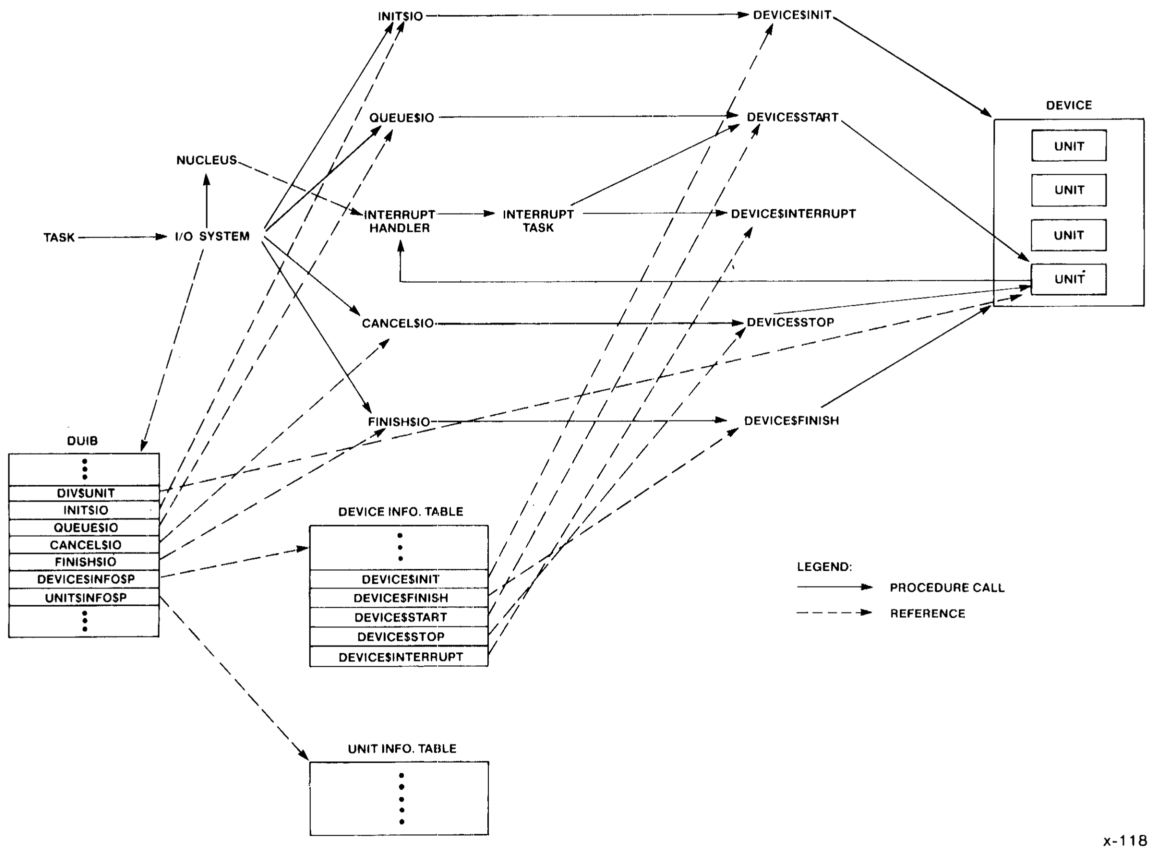


Figure 3-4. Relationships Between I/O Procedures and I/O Data Structures

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 Initialize I/O 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 Initialize I/O 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 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 204 devices use the same device driver code. The same Initialize I/O 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 incarnations of the routines that service units of a particular device are able to access the device data area for that device.

Although the common and random access device drivers already provide this mechanism, you may want to include a device data storage area in any custom driver that you write.

WRITING DRIVERS FOR USE WITH BOTH iRMX™ 86- AND iRMX™ 88-BASED SYSTEMS

A common or random access device driver that makes no system calls is compatible with both the iRMX 86 and iRMX 88 I/O Systems. Consequently, such a device driver can be "ported" between applications based on the two iRMX systems.

***

CHAPTER 4
I/O REQUESTS

This chapter contains two kinds of information that writers of drivers for devices other than terminals will find useful. Presented first are summaries of the actions that the I/O System takes in response to the various kinds of I/O requests that application tasks can make. Next are three tables -- one for each type of device driver -- that show which DUIB and IORS fields device drivers should be concerned with.

I/O SYSTEM RESPONSES TO I/O REQUESTS

This section shows which device driver procedures the I/O System calls when it processes each of the eight kinds of I/O requests. When there are multiple calls, the order of the calls is significant.

ATTACH DEVICE REQUESTS

When the I/O System receives the first attach device request for a device, it makes the following calls, in order, to device driver procedures:

The Call	The Effects of the Call
Initialize I/O	The driver resets the device as a whole and creates the device data storage area and interrupt task(s).
Queue I/O, with the FUNCT field of the IORS set to F$ATTACH (=4)	The driver resets the selected unit.

When the I/O System receives an attach device request that is not the first for the device, it makes the following call:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$ATTACH (=4)	The driver resets the selected unit.

DETACH DEVICE REQUESTS

When the I/O System receives a detach device request, and there is more than one unit of the device attached, it makes the following call:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$DETACH (=5)	The driver performs cleanup operations for the selected unit, if necessary.

When the I/O System receives a detach device request, and there is only one attached unit on the device, it makes the following calls, in order, to device driver procedures:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$DETACH (=5)	The driver performs cleanup operations for the selected unit, if necessary.
Finish I/O	The driver performs cleanup operations for the device as a whole (if necessary) and deletes the objects created by Initialize I/O.

READ, WRITE, OPEN, CLOSE, SEEK, AND SPECIAL REQUESTS

When the I/O System receives a read, write, open, close, seek, or special request, it makes the following call to a device driver procedure:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$READ (=0), F$WRITE (=1), F$OPEN (=6), F$CLOSE (=7), F$SEEK (=2), or F$SPECIAL (=3), depending on the type of the I/O request.	The driver performs the requested operation. (F$OPEN and F$CLOSE usually require no processing.)

CANCEL REQUESTS

When a connection is deleted while I/O might be in progress, such as when an iRMX 86 job is deleted, the I/O System makes the following calls, in order, to device driver procedures:

The Call	The Effects of the Call
Cancel I/O	The driver removes from the request queue all requests that contain the same Cancel ID value as that in the current request, and stops processing if necessary.
Queue I/O, 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 (exchange).

DUIB AND IORS FIELDS USED BY DEVICE DRIVERS

Tables 4-1, 4-2, and 4-3 indicate, for each type of device driver, the fields of DUIBs and IORSs with which user-written portions of device drivers need to be concerned.

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$status	w	w	w	w	w	w	w	w
Actual					w	w		
Actual$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								m
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 --- is available for any purpose suiting the needs of the device driver

***

CHAPTER 5
WRITING COMMON OR RANDOM ACCESS DEVICE DRIVERS

This chapter contains the calling sequences for the procedures that 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 to providing information about the procedures that common or random access drivers must supply, this chapter describes the purpose and calling sequence for each of five procedures, two of which random access device drivers in iRMX 86 applications must call under certain conditions.

INTRODUCTION TO PROCEDURES THAT DEVICE DRIVERS MUST SUPPLY

The routines that are provided by the I/O System and that the I/O System calls (INIT$IO, FINISH$IO, QUEUE$IO, CANCEL$IO, and INTERRUPT$TASK for iRMX 86 systems) (RAD$INIT$IO, RAD$FINISH$IO, RAD$QUEUE$IO, RAD$CANCEL$IO, and INTERRRUPT$TASK for iRMX 88 systems) constitute the bulk of a common or random access device driver. These routines, in turn, make calls to device-dependent routines that you 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 (the I/O System-supplied interrupt task) call this procedure.

A device stop procedure. 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 this procedure.

A device interrupt procedure. This procedure must do all of the device-dependent processing that results from the device sending an interrupt. INTERRUPT$TASK calls this procedure.

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. 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$p	POINTER to the user portion of the device's data storage area. You must specify the size of this portion in the Device Information Table for this device. The device initialization procedure can use this data area for whatever purposes it chooses. Possible uses for this data area include local flags and buffer areas.
status$p	POINTER to a WORD in which the device initialization procedure must return the status of the initialization operation. 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.

If you have a device that does not need to be initialized before it can be used, you can use the default device initialization procedure supplied by the I/O System. The name of this procedure is DEFAULT$INIT. Specify this name in the Device Information Table. DEFAULT$INIT does nothing but return the E$OK condition code.

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. 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. 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.

DEVICE START PROCEDURE

Both QUEUE$IO and INTERRUPT$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 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. 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 byte where I/O is to commence, and the buffer address.

duib$p	POINTER to the DUIB of the device-unit for which the I/O request is intended. The device start procedure can use the DUIB to access the Device Information Table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. 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 functions supported by the device and recognize that requests for nonsupported functions are error conditions.

- 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 an 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 field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 device driver returns 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.

- 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 device driver support routines wait until the device sends an interrupt and the device interrupt procedure sets IORS.DONE to TRUE, before determining that the request is actually finished.

DEVICE STOP PROCEDURE

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. The device stop procedure needs this information to determine what type of function to stop.
duib$p	POINTER to the DUIB of the device-unit on which the I/O function is being performed.
ddata$p	POINTER to the user portion of the device's data storage area. 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 and use the default procedure supplied with the I/O System. The name of this procedure is DEFAULT$STOP. Specify this name in the Device Information Table. DEFAULT$STOP simply returns to the caller.

DEVICE INTERRUPT PROCEDURE

INTERRUPT$TASK calls the user-written device interrupt procedure to process an interrupt that just occurred. 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. The device interrupt procedure must update information in this IORS. A value of zero 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.

ddata$p	POINTER to the user portion of the device's data storage area. The device interrupt procedure can update flags in this data area or retrieve data sent by the device.

The device interrupt procedure must do the following:

- It must determine whether the interrupt resulted from the completion of an I/O function by the correct device-unit.
- If the correct device-unit did send the interrupt, the device interrupt procedure must determine 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. This may involve setting flags in the user portion of the data storage area, tranferring 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 field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 and 206 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.
- If no error has occurred, it must set the IORS.STATUS field to indicate an E$OK condition.

PROCEDURES THAT iRMX™ 86 RANDOM ACCESS DRIVERS MUST CALL

There are several procedures that random access drivers in iRMX 86 applications can 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).

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 BASIC I/O SYSTEM REFERENCE 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.

SEEK$COMPLETE PROCEDURE

In most applications, it is desirable to overlap seek operations (which can take relatively long periods of time) with other operations. 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 (OFFH).

   - Some devices send only one interrupt 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 (OFFH) immediately.

   - Some devices send two interrupts 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 from the device arrives, the device interrupt procedure sets the DONE flag to TRUE (OFFH).

3. When the interrupt 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 the driver does not call SEEK$COMPLETE, then CYLINDER$SIZE must be configured to zero.

PROCEDURES FOR OTHER LONG-TERM OPERATIONS

The iRMX 86 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).

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 (OFFH).

   - Some devices send only one interrupt 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 (OFFH) immediately.

   - Some devices send two interrupts 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 from the device arrives, the device interrupt procedure sets the DONE flag to TRUE (OFFH).

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 which 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 passes 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 to indicate the end of the long-term operation.

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 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 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 passes to the device start and device interrupt procedures.

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$IORS allows your driver procedure to obtain the token of the IORS for the previous long-term request, so that it can modify the IORS to initiate new I/O requests. The IORS$P that INTERRUPT$TASK passed to the device interrupt procedure is set to zero (for units busy performing a seek or other long-term operation). Therefore, the driver can only access the IORS in this manner.

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 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 (0).

4. The device interrupt procedure calls END$LONG$TERM$OPERATION.

The syntax of the call to GET$IORS is as follows:

```
iors$base = GET$IORS(unit, ddata$p);
```

where:

iors$base	SELECTOR in which the random access support routines return the base portion of the IORS. Use the PL/M-86 built-in procedure BUILD$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 passes to the device start and device interrupt procedures.

FORMATTING CONSIDERATIONS

If you write a random access driver and you intend to use the Human Interface FORMAT command (for iRMX 86 systems) or the RQ$FORMAT call (for iRMX 88 systems) 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.

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.

***

CHAPTER 6 WRITING A CUSTOM DEVICE DRIVER

Custom device drivers are drivers that you create in their entirety because your device doesn't fit into either the common or random access 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.

In order for the I/O System to communicate with your device driver procedures, you must provide the addresses of these four procedures for 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.

INITIALIZE I/O PROCEDURE

The iRMX 86 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 iRMX 88 I/O System calls the Initialize I/O procedure when an application task attaches or creates a file on the device and no other files on the device are currently attached. In either case, the I/O System calls the Initialize I/O procedure before calling any other driver procedure.

The Initialize I/O procedure must perform any initial processing necessary for the device or the driver. If the device requires an interrupt task (or region or device data area, in the case of iRMX 86 drivers), the Initialize I/O procedure should create it (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. The init$io procedure uses this DUIB to determine the characteristics of the unit.
ddata$p	POINTER to a WORD 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$io procedure should return a zero via this pointer.
status$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.

FINISH I/O PROCEDURE

The iRMX 86 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 iRMX 88 I/O System calls the Finish I/O procedure when an application task detaches or deletes the last remaining file connection for the 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$io	Name of the Finish I/O procedure. You can specify any name for this procedure as long as it does not conflict with other procedure names. You must, however, provide its starting address in the DUIBs of all device-units that it services.
duib$p	POINTER to the DUIB of the device-unit of the device being detached. The finish$io procedure needs this DUIB in order to determine the device on which to perform the final processing.
ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure. The finish$io procedure must delete this resource and any others created by driver routines.

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$io	Name of the Queue I/O procedure. You can use any name for this procedure as long as it does not conflict with other procedure names. You must, however, provide its starting address for the DUIBs of all device-units that it services.
iors$t	SELECTOR containing the location of an IORS. This IORS describes the request. When the request is processed, the driver (though not necessarily the queue$io procedure) must fill in the status fields and send the IORS to the response mailbox (exchange) indicated in the IORS. Chapter 2 describes the format of the IORS. It lists the information that the I/O System supplies when it passes the IORS to the queue$io procedure and indicates the fields of the IORS that the device driver must fill in.

duib$p	POINTER to the DUIB of the device-unit for which the request is intended.
ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure. The queue$io procedure can place any necessary information in this area in order to update the request queue or status fields.

CANCEL I/O PROCEDURE

The I/O System can call the Cancel I/O procedure in order to cancel one or more previously queued I/O requests. The iRMX 88 I/O System does not call Cancel I/O, but in the iRMX 86 environment Cancel I/O is called under either of the following two conditions:

- If the user makes an RQ$A$PHYSICAL$DETACH$DEVICE system call and specifies the hard detach option (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE 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 device cannot guarantee that a request will be finished within a fixed amount of time (such as waiting for input from a terminal keyboard), 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$id	Name of the Cancel I/O procedure. You can use any name for this procedure as long as it doesn't conflict with other procedure names. You must, however, provide its starting address in the DUIBs of all device-units that it services.
cancel$id	WORD containing the id value for the I/O requests that are are to be cancelled. Any pending requests with this value in the cancel$id field of their IORS's must be removed from the queue of requests by the Cancel I/O procedure. Moreover, the I/O System places a CLOSE request with the same cancel$id value in the queue. The CLOSE request must not be processed until all other requests with that cancel$id value have been returned to the I/O System.

duib$p	POINTER to the DUIB of the device-unit for which the request cancellation is intended.
ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure. This area may contain the request queue.

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. The common and random access device drivers form 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, common and random access device drivers implement 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 common and random access driver Queue I/O procedures make 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 is actually sending 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 back to the last IORS in the queue. A queue of this sort is illustrated in Figure 6-1.

The device driver can add or remove requests from the queue by adjusting LINK$FOR and LINK$BACK pointers in the IORSs.

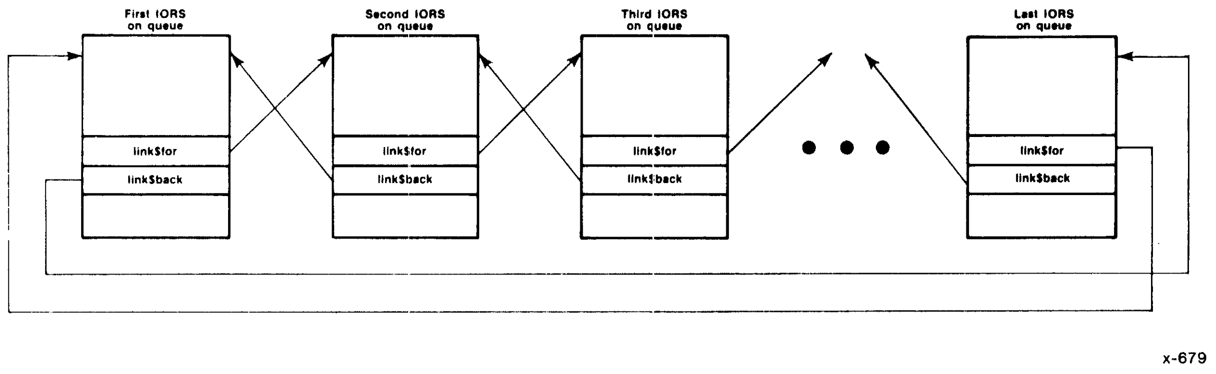


Figure 6-1. Request Queue

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 0. Otherwise, head$queue contains the address of the first IORS in the queue.

***

CHAPTER 7 TERMINAL DRIVERS

Both the iRMX 86 and iRMX 88 Operating Systems supply a Terminal Handler that can serve as an interface between the Nucleus and a terminal device. This interface is minimal and allows limited interaction between the terminal operator and the Operating System. However, the iRMX 86 Operating System also provides an interface to terminals via the Basic I/O System. This interface allows tasks to use the power and convenience of I/O System calls when communicating with terminals. To add support for new terminal controllers in the Basic I/O System, you can write device drivers, which provide the software link between the Operating System software (called the Terminal Support Code) and the terminal.

The iRMX 88 Executive does not support terminal drivers as outlined in this chapter.

This chapter explains how to write a terminal driver whose capabilities include handling single-character I/O, parity checking, answering and hanging up functions on a modem, and automatic baud rate searching for each of several terminals. Such a driver is neither common, random access, nor custom. Consequently, this chapter is more self-contained than Chapters 5 and 6; it describes the data structures used by terminal drivers, as well as the procedures that you must provide.

TERMINAL SUPPORT CODE

As in the case of common and random access drivers, the I/O System provides the procedures that the I/O System invokes when performing terminal I/O. They are known collectively as the Terminal Support Code. Figure 7-1 shows schematically the relationships between the various layers of code that are involved in driving a terminal.

Among the duties performed by the Terminal Support Code are managing buffers and maintaining several terminal-related modes.

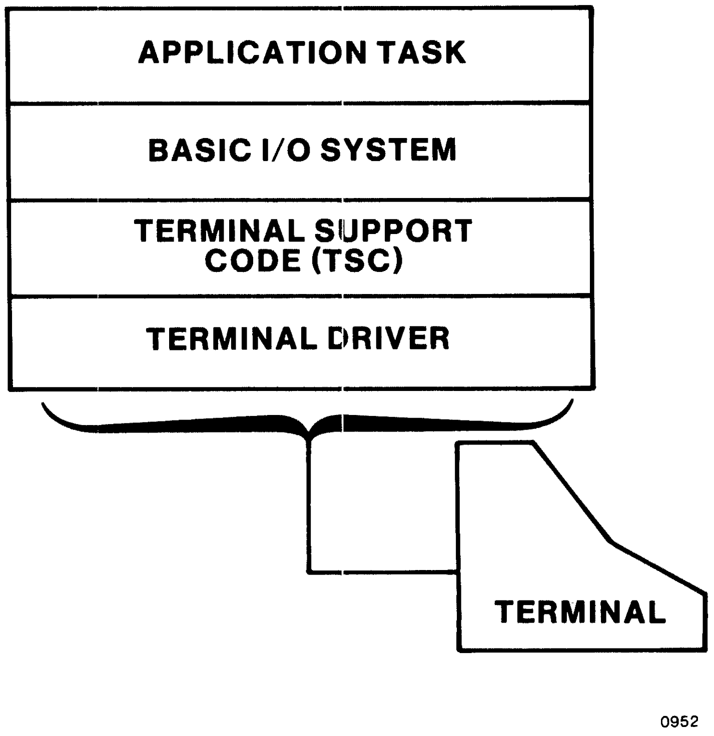


Figure 7-1. Software Layers Supporting Terminal I/O

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.

DUIB

This section lists the elements that make up a DUIB for a device-unit that is a terminal. When creating DUIBs for iRMX 86 applications, code them in the format shown here (as assembly-language structures). If you give the iRMX 86 ICU the pathname of your Unit Information Table field, the iRMX 86 Interactive Configuration Utility (ICU) includes your DUIB file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the 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
&   >
```

DEVICE INFORMATION TABLE

A terminal's Device Information Table provides information about a terminal controller. When creating these tables, code them in the format shown here (as assembly-language declarations). If you give the iRMX 86 ICU the pathname of your Unit Information Table field, the ICU includes the file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file).

The fields TERM$INIT, TERM$FINISH, TERM$SETUP, TERM$OUT, TERM$ANSWER, TERM$HANGUP, and TERM$CHECK contain the names of user-supplied procedures whose duties are described later in this chapter. When creating the file containing your Device Information Tables, specify external declarations for these user-supplied procedures. This allows the code for these user-supplied procedures to be included in the generation of the I/O System. For example, if your procedures are named TERM$INIT, TERM$FINISH, TERM$SETUP, TERM$OUT, TERM$ANSWER, TERM$HANGUP, and TERM$CHECK, include the following declarations in the file containing your Device Information Tables:

```
extrn term$init: near
extrn term$finish: near
extrn term$setup: near
extrn term$out: near
extrn term$answer: near
extrn term$hangup: near
extrn term$check: near
```

Use the following 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    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
                    .
                    .
                    .
```

where:

NUM$UNITS	WORD containing the number of terminals on this terminal controller.
DRIVER$DATA$SIZE	WORD containing the number of bytes in the driver's data area pointed to by the USER$DATA$PTR field of the TSC Data structure.
STACK$SIZE	WORD containing the number of bytes of stack needed collectively by the user-supplied procedures in this device driver.
TERM$INIT	WORD specifying the address of this controller's user-written terminal initialization procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$FINISH	WORD specifying 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	WORD specifying 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$OUT	WORD specifying the address of this controller's user-written terminal output procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$ANSWER	WORD specifying 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$HANGUP	WORD specifying the address of this controller's user-written terminal hangup procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
NUM$INTERRUPTS	WORD containing the number of interrupt lines that this controller uses. You must define an INTERRUPT$LEVEL and TERM$CHECK word for each interrupt.
INTERRUPT$LEVEL	WORDs containing the level numbers of the interrupts that are associated with the terminals driven by this controller. You must supply one such word for each interrupt the controller uses.
TERM$CHECK	WORDs specifying 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$OUT to be called.
DRIVER$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 10 + 2*(number of interrupt levels used by the driver)

You must supply the TERM$INIT, TERM$FINISH, TERM$SETUP, TERM$OUT, TERM$ANSWER, TERM$HANGUP, and TERM$CHECK procedures. However, if your terminals are not used with modems, the TERM$ANSWER and TERM$HANGUP procedures can simply contain a RETURN. Also, if your application does not need to perform special processing when all of the terminals on the controller are detached, the TERM$FINISH procedure also can simply contain a RETURN.

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). When creating Unit Information Tables, code them in the format shown here (as assembly-language declarations). If you give the iRMX 86 ICU; the pathname of your Unit Information Table field, the ICU includes the file in the assemgly of IDEVCF.A86 (a Basic I/O System configuration file).

```
TERMINAL$UNIT$INFORMATION
     DW   CONN$FLAGS
     DW   TERM$FLAGS
     DW   IN$RATE
     DW   OUT$RATE
     DW   SCROLL$NUMBER
     DW   FLOW$CONTROL*
     DW   HIGH$WATER$MARK*
     DW   LOW$WATER$MARK*
     DW   FC$ON$CHAR*
     DW   FC$OFF$CHAR*
```

*These elements apply only to buffered device drivers and are useful only if you must specify them at configuration time.

where:

CONN$FLAGS — WORD specifying the default connection flags for this terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE 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 line editing (flush mode).
2	Echo control. 0 = Echo. 1 = Do not echo.
3	Input parity control. 0 = Set parity bit to 0. 1 = Do not alter parity bit.
4	Output parity control. 0 = Set parity bit to 0. 1 = Do not alter parity bit.

Bits	Value and Meaning
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. 3 = Do not act upon any OSC sequences.
8-15	Reserved bits. For future compatibility, set to 0.

TERM$FLAGS

WORD specifying the terminal connection flags for this terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE 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.

Bits	Value and Meaning
4-5	Input parity control.
	0 = Always set parity bit to 0.
	1 = Never alter the parity bit.
	2 = Even parity is expected on input. Set the parity bit to 0 unless the received byte has odd parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.)
	3 = Odd parity is expected in input. Set the parity bit to 0 unless the received byte has even parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.)
6-8	Output parity control.
	0 = Always set parity bit to 0.
	1 = Always set parity bit to 1.
	2 = Set parity bit to give the byte even parity.
	3 = Set parity bit to give the byte odd parity.
	4 = Do not alter the parity bit.
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.

Bits	Value and Meaning
	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.

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).

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 =	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 (Control-W is the default).

The Unit Information Table can contain additional data, depending on the needs of the controller. Refer to the "Additional Information for Buffered Devices" section of this chapter for information about other fields you can add to the table.

TERMINAL SUPPORT CODE (TSC) DATA AREA

DUIBs, Device Information Tables, and Unit Information Tables are structures that you set up at configuration time to provide information about the initial state of your terminals. During configuration, the ICU assembles these tables into the code segment of the Basic I/O System. 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$UNITS field in the Device Information Table specifies the number of unit portions that the Basic I/O System creates.
- A user portion which the user-written driver routines can use in any manner they choose. The DRIVER$DATA$SIZE field in the Device Information Table specifies the length of this portion. One of the fields in the controller portion (USER$DATA$PTR) points to the beginning of this field.

Figure 7-2 illustrates the TSC Data Area graphically.

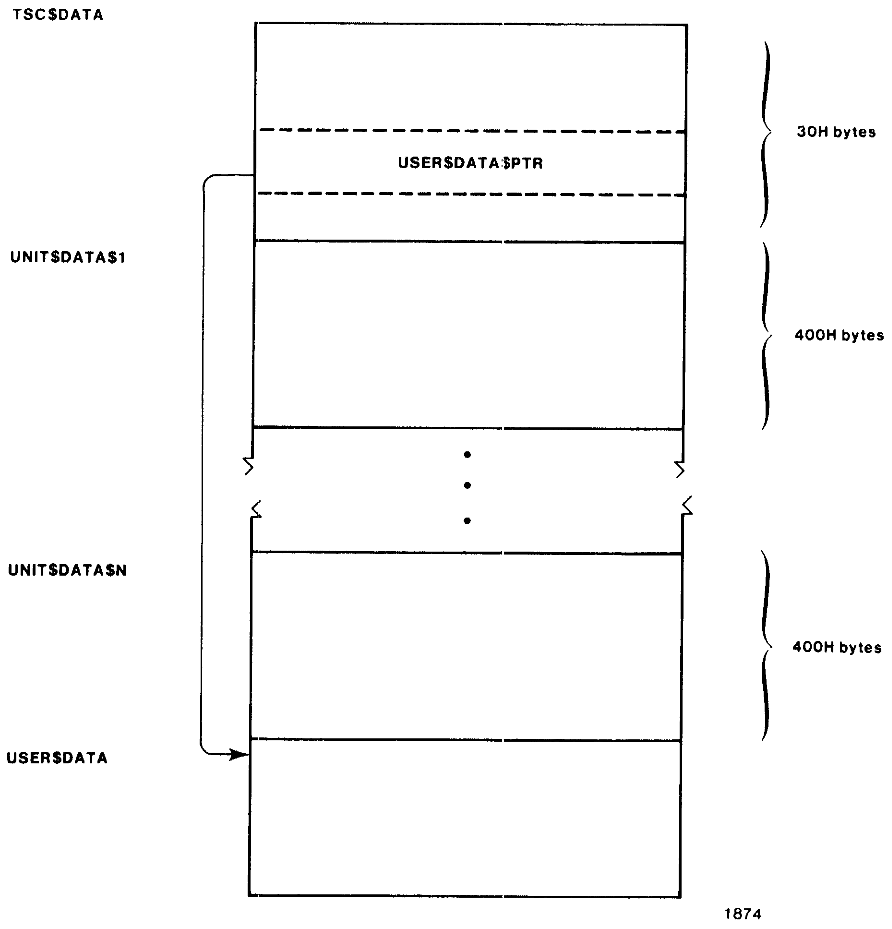


Figure 7-2. TSC Data Area

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.

The TSC Data Area always starts on a segment boundary Its structure is 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,
     RESERVED1(901)                 BYTE,
     BUFFERED$DEVICE$DATA(11)       BYTE,
     RESERVED2(100)                 BYTE);
```

where:

IOS$DATA$SEGMENT — SELECTOR containing the base address of the I/O System's data segment. The I/O System's terminal support routine TSINITIO fills in this information during initialization.

STATUS — WORD in which the user-written terminal initialization procedure must return status information.

INTERRUPT$TYPE — BYTE in which the user-written terminal check procedure must return the encoded interrupt type. The possible values are:

0	None
1	Input interrupt
2	Output interrupt
3	Ring interrupt
4	Carrier interrupt
5	Delay interrupt

If the terminal check procedure detects that there are more interrupts to service, the terminal check procedure adds the following value:

8	More interrupts

to the encoded interrupt type it returns.

For more information about these codes and their values, see the description of the terminal check procedure in the next section.

INTERRUPTING$UNIT — 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	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	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.
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	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	WORD specifying the connection flags for this terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE 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.

Bits	Value and Meaning
4-5	Input parity control. 0 = Always set parity bit (bit 7) to 0. 1 = Never alter the parity bit. 2 = Even parity is expected on input. Set the parity bit to 0 unless the received byte has odd parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.) 3 = Odd parity is expected in input. Set the parity bit to 0 unless the received byte has even parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.)
6-8	Output parity control. 0 = Always set parity bit to 0. 1 = Always set parity bit to 1. 2 = Set parity bit to give the byte even parity. 3 = Set parity bit to give the byte odd parity. 4 = Do not alter the parity bit.
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.

Bits	Value and Meaning
	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.

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).

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 =	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$NUMBER	WORD specifying the number of lines that are to be sent to the terminal each time the operator enters the appropriate control character (Control-W is the default).
BUFFERED$DEVICE$-DATA	BYTES that contain 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.

PROCEDURES THAT 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 task call 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 task call 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 task call 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 calls this procedure.

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 set output waiting procedure. This procedure signals the Terminal Support Code that a terminal is ready to perform character transmission and interrupt handling.

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 of the terminals connected to the controller, the Terminal Support Code passes a pointer to the beginning of the TSC Data Area (the device portion). On the other hand, 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 always starts on a paragraph boundary, 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-86 builtin procedure BUILD$PTR using the base part of the pointer it received and an offset of 0. Also, if a procedure, such as term$check, 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)]
```

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.)

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.

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. 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.

TERMINAL SETUP PROCEDURE

This procedure "sets up" one terminal according to the TERMINAL$FLAGS, IN$RATE, OUT$RATE, SCROLL$NUMBER, and BUFFERED$DEVICE$DATA fields in the corresponding UNIT$DATA portion of the TSC Data Area. In particular, if IN$RATE is 1, then the term$setup procedure must start a baud rate search. (The terminal check procedure usually finishes the search and then fills in IN$RATE with the actual baud rate.) If OUT$RATE is 0, the terminal setup procedure assumes the output baud rate is the same value as the input baud rate.

If your terminal controller is a buffered device (an intelligent device that manages its own internal data buffers), the terminal setup procedure must also set one of the reserved fields of the UNIT$DATA structure. Refer to the "Buffered Devices" section in this chapter for more information.

If your terminal driver supports a modem, the terminal setup procedure might have to perform additional services. Refer to the "Terminal Hangup" section for more information.

The terminal setup procedure must call the set output waiting procedure. Refer to a later section in this chapter for more information on the set output waiting procedure. 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.

TERMINAL ANSWER PROCEDURE

This procedure activates the Data Terminal Ready line for a particular terminal. The Terminal Support Code calls the terminal answer procedure only when <u>both</u> 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 the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL 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.

TERMINAL HANGUP PROCEDURE

This procedure clears the Data Terminal Ready 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 the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL 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 and then reactivate it.

TERMINAL CHECK PROCEDURE

The Terminal Support Code calls this procedure whenever an interrupt occurs, which usually signals 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.
2. If no input character is available, check for a transmitter ready to send another character.
3. If no transmit 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$UNIT field of the TSC Data Area and information about the type of interrupt in the INTERRUPT$TYPE field. The Terminal Support Code interprets values in the INTERRUPT$TYPE field as follows:

0	no interrupt
1	input interrupt
2	output interrrupt
3	ring interrupt
4	carrier interrupt
5	delay interrupt

Also, if the terminal check procedure detects another interrupt while it is returning information about the first interrupt, it should add the following value:

8	more interrupts

to the value it places in the INTERRUPT$TYPE field. 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:

OH	no interrupt
9H	input interrupt plus more
OAH	output interrupt plus more
OBH	ring interrupt plus more
OCH	carrier interrupt plus more
ODH	delay interrupt plus more

By returning these values, the terminal check procedure ensures that the Terminal Support Code calls it again. Otherwise, the driver could lose characters. If, in fact, 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.

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 in Appendix B to see how to implement a baud rate scan.

5. Place a value of ODH in the INTERRUPT$TYPE field (delay interrupt plus more). The ODH 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.

If the terminal check procedure encounters an input interrupt, it must also return the input character to the procedure that called it, adjusting the parity bit according to bits 4 and 5 of the TERMINAL$FLAGS field in the interrupting unit's UNIT$DATA structure. If the interrupt is not an input interrupt, the terminal check procedure can return any value.

The syntax of the call to the user-written terminal check procedure is as follows:

```
    input$char = term$check(tsc$data$ptr)
```

where:

input$char	BYTE in which the terminal check procedure returns the input character, if the interrupt was an input interrupt. If the interrupt was not an input interrupt, this parameter can have any value.

term$check	Name of the terminal check procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
tsc$data$ptr	POINTER to the start of the Terminal Support Code Data Area.

TERMINAL OUTPUT PROCEDURE

The Terminal Support Code calls this procedure to display a character at a terminal. 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.

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.

SET OUTPUT WAITING PROCEDURE

This procedure notifys the Terminal Support Code that the particular terminal is ready to perform data transmission.

The syntax of a call to the set output waiting procedure is as follows:

```
    CALL xts$set$output$waiting (unit$data$n$p);
```

where:

xts$set$output $waiting	Name of the Terminal Support Code provided procedure. The terminal setup procedure that you write must declare xts$set$output$waiting as an external procedure with one pointer parameter.
unit$data$n$ptr	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area, This si the same pointer passed to the terminal setup procedure by the Terminal Support Code.

ADDITIONAL INFORMATION FOR BUFFERED DEVICES

If you are writing a driver for a buffered communications device (an intelligent communications processor like the iSBC 544 board that manages its own buffers of data separately from the ones managed by the Terminal Support Code), 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 11 bytes:

```
DECLARE  BUFFERED$DEVICE$DATA  STRUCTURE(
    BUFFERED$DEVICE       BYTE,
    FLOW$CONTROL          WORD,
    HIGH$WATER$MARK       WORD,
    LOW$WATER$MARK        WORD,
    FC$ON$CHAR            WORD,
    FC$OFF$CHAR           WORD);
```

where:

BUFFERED$DEVICE — When true, a BYTE that specifies whether the unit requires handling as a buffered device.

FLOW$CONTROL — WORD specifying whether the communications board sends flow control characters (selected by the FC$ON$CHAR and FC$OFF$CHAR fields, but usually XON and XOFF) to turn input on and off. The low-order bit (bit 0) controls this option, 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.

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.

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.
FC$ON$CHAR	WORD specifying an ASCII character that the communication board sends to the connecting device when the number of bytes in its buffer drops to the low-water mark. Normally this character tells the connecting device to resume sending data.
FC$OFF$CHAR	A WORD specifying an ASCII character that the communication board sends to the connecting device when the number of characters in its buffer rises to the high-water mark. Normally this character tells the connecting device to stop sending data.

When a user attaches a unit on any terminal device, the Terminal Support Code calls the terminal setup procedure. If the device is a buffered device, the terminal setup procedure must set the BUFFERED$DEVICE field to TRUE (OFFH). It should also fill in the other fields of the BUFFERED$DEVICE$DATA structure. 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 (OH) 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" and a "detach device", the terminal setup procedure should establish its own internal flags (one for each unit) in addition to the BUFFERED$DEVICE fields. It can use these flags 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 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."

PROCEDURES' USE OF DATA STRUCTURES

Table 7-1 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

Also, "System" and "ICU" are used in Table 7-1 to indicate the iRMX 86 software and the iRMX 86 Interactive Configuration Utility, respectively. In addition, "Term$flags" is an abbreviation of "Terminal$flags," and numbers following immediately after "Term$flags" are bit numbers in that word.

Table 7-1. Uses of Fields in Terminal Driver Data Structures

	Filled in/Changed by	Can or Will be Used by
TSC$DATA		
IOS$DATA$SEGMENT	System	(1)-(7)
STATUS	(1)	System
INTERRUPT$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	System
TERM$FLAGS (0-2)	System	System
TERM$FLAGS (3)	System	(3)
TERM$FLAGS (4-5)	System	(3),(6)
TERM$FLAGS (6-8)	System	(3),(6),(7)
IN$RATE	System,(3),(6)	(3)
OUT$RATE	System	(3)
SCROLL$NUMBER	System	System
BUFFERED$DEVICE$DATA	(3)	System, (3)
TERMINAL$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$HANGUP	ICU	System
TERM$CHECK	ICU	System
INTERRUPTS		
INTERRUPT$LEVEL	ICU	System
TERM$CHECK	ICU	System
DRIVER$INFO	ICU	(1)-(7)

***

CHAPTER 8 BINDING A DEVICE DRIVER TO THE I/O SYSTEM

You can write the modules for your device driver in either PL/M-86 or the ASM86 Macro Assembly Language. However, you must adhere to the following guidelines:

- If you use PL/M-86, 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-86 COMPACT size control. In particular, your routines must function in the same manner as reentrant PL/M-86 procedures with the ROM and COMPACT controls set. The ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS manual describes these conditions and conventions.

USING THE iRMX™ 86 INTERACTIVE CONFIGURATION UTILITY

To use the iRMX 86 Interactive Configuration Utility to configure a driver that you have written for your application system, you must perform the following steps:

1. For each device driver that you have written, assemble or compile the code for the driver.

2. Put all the resulting object modules in a single library, such as DRIVER.LIB.

3. Ascertain the device numbers and device-unit numbers to use in the DUIBs for your devices.

   a. Use the ICU to configure a system containing all the Intel-supplied drivers you require.

   b. Use the G option to generate that system.

   c. Use a text editor to examine the file IDEVCF.A86. Among other things, this file contains DUIBs for all the device-units you defined in your configuration.

   d. Look for the DEFINE_DUIB structures in the file. Chapter 2 lists the format of these structures. Note the device number (eighth field) and the device-unit number (tenth field) of the last DUIB defined in the file.

   Figure 8-1 lists part of an IDEVCF.86 file which contains this information (the file you examine might look different, depending on how you configure your system). The arrows in the figure point to the relevant fields.

e. Use the next available device numbers and device-unit numbers in your DUIBs.

```
          .
          .
          .
     DEFINEDUIB <
     & 'lp',
     & 00001H,
     & 0F2H,
     & 00,
     & 00,
     & 00,
     & 00,
---> & 00004H,
     & 00,
---> & 0000BH,
     & INITIO,
     & FINISHIO,
     & QUEUEIO,
     & CANCELIO,
     & DINFO04,
     & 00,
     & 0FFFFH,
     & 00000H,
     & 130,
     & FALSE,
     & 00000H,
     & 0
     &>
     NUMDUIB EQU (THIS BYTE - DUIBTABLE) / SIZE DEFINEDUIB
     BIOSCODE ENDS
     %DEVICETABLES(NUMDUIB,0000CH,005H,003E8H)
     CODE SEGMENT
     ASSUME CS:CGROUP
          .
          .
          .
```

Figure 8-1. Example IDEVCF.A86 File

4. 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 2. Place all the structures in the same file. Later, the ICU includes this file in the assembly of the IDEVCF.A86 file.

   b. A file containing all the device information tables you are adding. Use the RADEV_DEV_INFO structures shown in Chapter 2 for any random access drivers you add. Later, the ICU includes this file in the assembly of the IDEVCF.A86 file.

   c. If applicable, any unit information table(s). Use the RADEV_UNIT_INFO structures shown in Chapter 2 for any random access drivers you add. Add these tables to the file created in step b.

   d. 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.

5. 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 (this means drivers that you have written).

   b. As input to the "User Devices" screen, enter the pathname of your device driver library. This refers to the library built in step 2; for example, :F1:DRIVER.LIB.

   c. Also, enter the information the ICU needs to include your configuration data in the assembly of IDEVCF.A86. The information needed includes the following:

      - DUIB source code pathname (the file created in step 4a).

      - Device and Unit source code pathname (the file created in steps 4b through 4d).

      - Number of user defined devices.

      - Number of user defined device-units.

The ICU does the rest.

Figure 8-2 contains an example of the "User Devices" screen. The underlined text represents user input to the ICU. In this example, the file :F1:DRIVER.LIB contains the object code for the driver, :F1:DUIB.SRC contains the source code for the DUIBs, and :F1:DEVINF.SRC contains the source code for the Device and Unit Information Tables along with the necessary external procedure declarations.

The code in the DRIVER.LIB file supports one device with two units. Refer to the iRMX 86 CONFIGURATION GUIDE for instructions on how to use the ICU.

```
User Devices
(OPN) Object Code Path Name [1-45 characters]
                            NONE
(DPN) Duib Source Code Path Name [1-45 characters]

(DUP) Device and Unit Source Code Path Name [1-45 characters]

(ND)  Number of User Defined Devices [0-0FFH]          0001H
(NDU) Number of User Defined Device-Units [0-0FFH]     0001H

Enter Changes [Abbreviations ?/= new_value] : OPN = :F1:DRIVER.LIB
: DPN = :F1:DUIB.SRC
: DUP = :F1:DEVINF.SRC
: ND = 1
: NDU = 2
```

Figure 8-2. Example User Devices Screen

USING THE iRMX™ 88 INTERACTIVE CONFIGURATION UTILITY

To use the iRMX 88 Interactive Configuration Utility to configure a driver that you have written for your application system, you must perform the following steps in the following order:

1. For each driver, assemble or compile the code.

2. When using the ICU:

   a. Answer "208", "215", "common", "random", or "custom" when asked for device type.

   b. When prompted, enter the information for the DUIBs, the device information tables, and, if applicable, the unit information table.

   c. When prompted for linking information, enter the names of the appropriate modules.

The ICU does the rest.

***

APPENDIX A RANDOM ACCESS DRIVER SUPPORT ROUTINES

This appendix describes, in general terms, the operations of the random access device driver support routines. The routines described include:

INIT$IO
FINISH$IO
QUEUE$IO
CANCEL$IO
INTERRUPT$TASK

NOTE

For iRMX 88 systems, these names are prefixed by "RAD$".

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 user-written 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 (the random access driver support routines follow essentially the same pattern). Or you can use it as a guideline for writing custom device drivers.

INIT$IO PROCEDURE

The iRMX 86 I/O System calls INIT$IO when an application task makes an RQ$A$PHYSICAL$ATTACH$DEVICE system call and there are no units of the device currently attached. The iRMX 88 I/O System calls INIT$IO when an application task attaches or creates a file on the device and no other files on the device are 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
- In the iRMX 86 environment, 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 that the INIT$IO procedure goes through in order to initialize the device. Figure A-1 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

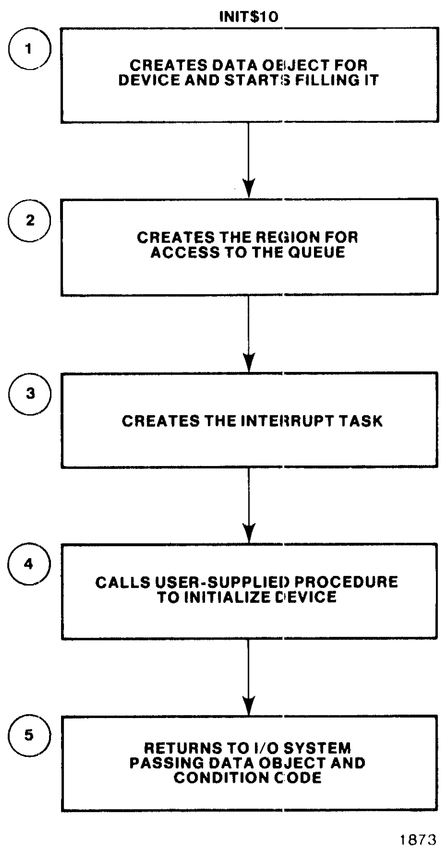

INIT$IO
1
CREATES DATA OBJECT FOR DEVICE AND STARTS FILLING IT
2
CREATES THE REGION FOR ACCESS TO THE QUEUE
3
CREATES THE INTERRUPT TASK
4
CALLS USER-SUPPLIED PROCEDURE TO INITIALIZE DEVICE
5
RETURNS TO I/O SYSTEM PASSING DATA OBJECT AND CONDITION CODE
1873

Figure A-1. Random Access Device Driver Initialize I/O Procedure

1. It creates a data storage area that will be used by all of 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:

   - The value of the DS (data segment) register.
   - A token (identifier) for a region (exchange) --- for mutual exclusion.
   - An array which will 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 to this array.
   - A token (identifier) for the interrupt task.
   - Other values indicating that 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 procedures of the device driver 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 creates an interrupt task to handle interrupts generated by this device. INIT$IO passes to the interrupt task a token for the data storage area. This area is where the interrupt task will get information about the device. Also, INIT$IO places a token for the interrupt task in the data storage area.

4. It calls a user-written 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 3 for information on how to write this initialization procedure.

5. 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 initialize operation.

FINISH$IO PROCEDURE

The iRMX 86 I/O System calls FINISH$IO when an application task makes an RQ$A$PHYSICAL$DETACH$DEVICE system call and there are no other units of the device currently attached. The iRMX 88 I/O System calls FINISH$IO when an application detaches or deletes a file and no other files on the device are attached.

FINISH$IO deletes the objects used by the other device driver routines, deletes the interrupt task, and calls a user-supplied procedure to perform final processing on the device itself.

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 selector to 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 user-written 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 4 for information about device information tables.

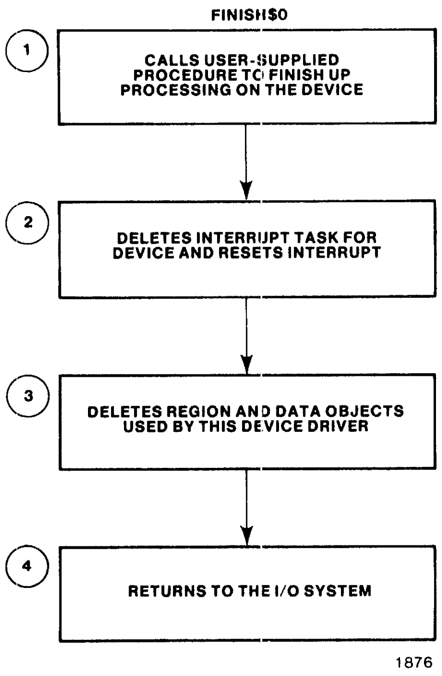

FINISH$O
1
CALLS USER-SUPPLIED PROCEDURE TO FINISH UP PROCESSING ON THE DEVICE
2
DELETES INTERRUPT TASK FOR DEVICE AND RESETS INTERRUPT
3
DELETES REGION AND DATA OBJECTS USED BY THIS DEVICE DRIVER
4
RETURNS TO THE I/O SYSTEM
1876

Figure A-2. Random Access Device Driver Finish I/O Procedure

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.

QUEUE$IO PROCEDURE

The I/O System calls the QUEUE$IO procedure to place an I/O request on a queue of requests. This queue has the structure of the doubly-linked list shown in Figure 2-2. If the device itself is not busy, QUEUE$IO also starts the request.

When the I/O System calls QUEUE$IO, it passes the following parameters

- A token (identifier) for the IORS
- A pointer to the DUIB
- A token (identifier) 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 that 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.

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 places the IORS on the queue.

4. It calls an I/O System-supplied procedure to start the processing of the request at the head of the queue. This results in a call to a user-written device start procedure which actually sends the data to the device itself. This start procedure is described in Chapter 5. If the device is already busy processing some other request, this step does not start the data transfer.

5. It surrenders control of the region, thus allowing other routines to have access to the queue.

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 iRMX 86 I/O System calls this procedure in one of two instances:

- If an iRMX 86 user makes an RQ$A$PHYSICAL$DETACH$DEVICE system call and specifies the hard detach option (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for information about this system call). The hard detach removes all requests from the queue.

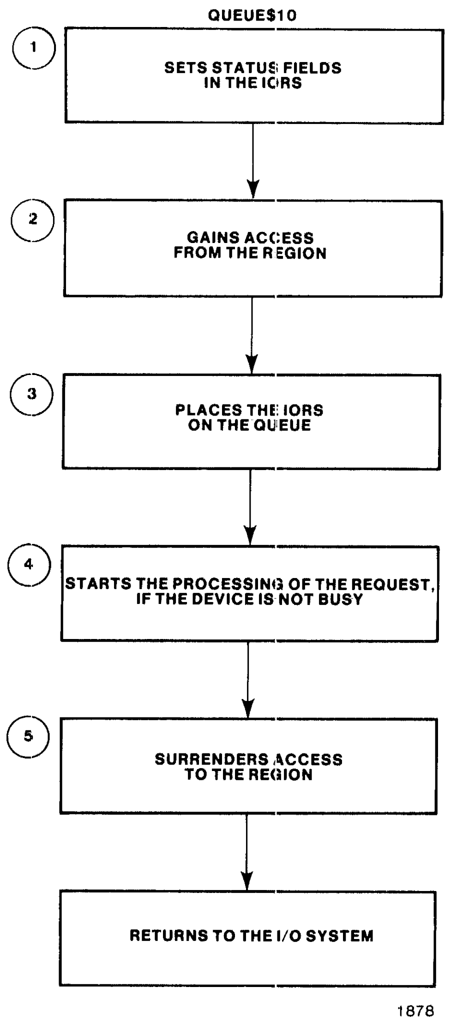

QUEUE$IO
1
SETS STATUS FIELDS IN THE IORS
2
GAINS ACCESS FROM THE REGION
3
PLACES THE IORS ON THE QUEUE
4
STARTS THE PROCESSING OF THE REQUEST, IF THE DEVICE IS NOT BUSY
5
SURRENDERS ACCESS TO THE REGION
RETURNS TO THE I/O SYSTEM
1878

Figure A-3. Random Access Device Driver Queue I/O Procedure

- 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 cancelled
- A pointer to the DUIB
- A token (identifier) 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 a request that is to be cancelled 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 cancelled is at the head of the queue, that is, the device is processing the request, CANCEL$IO calls a user-written 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 (exchange) indicated in the IORS.

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.

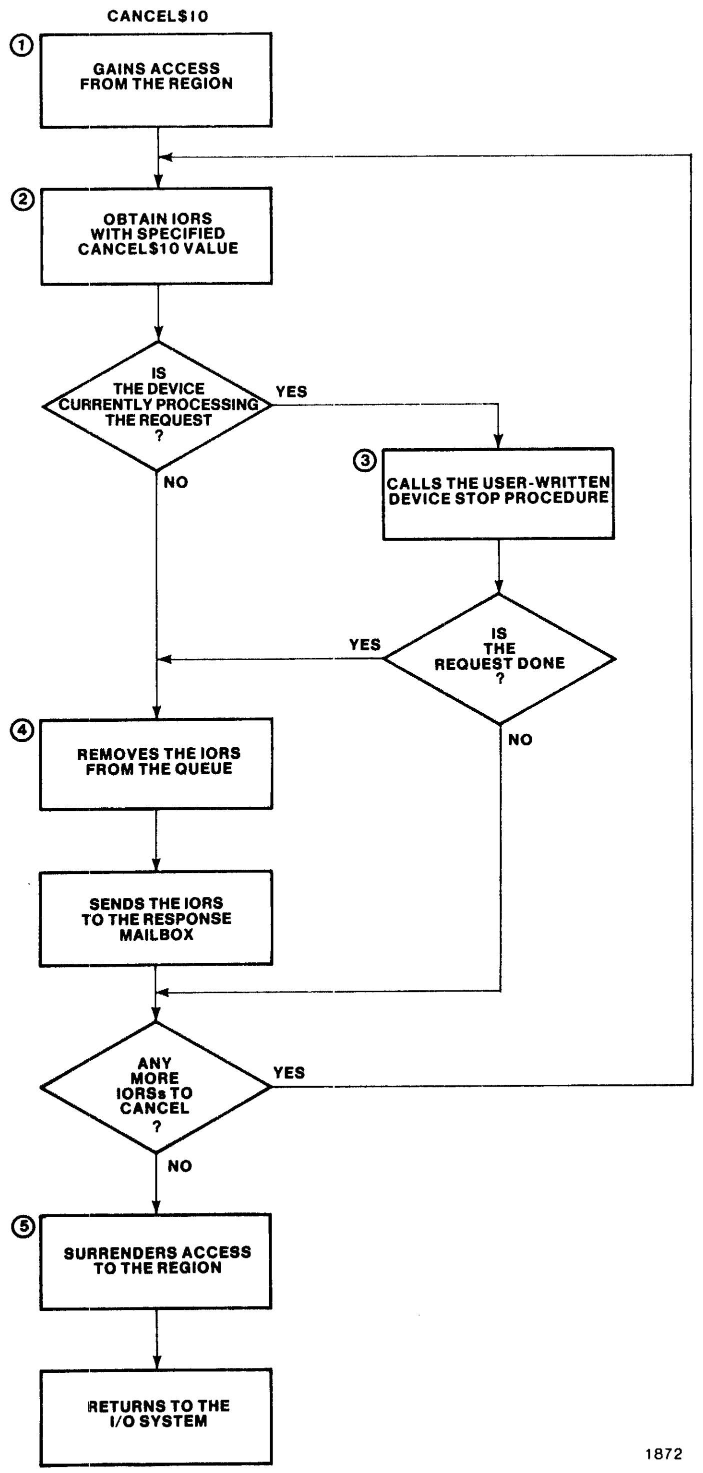

CANCEL$10
1
GAINS ACCESS FROM THE REGION
2
OBTAIN IORS WITH SPECIFIED CANCEL$10 VALUE
IS THE DEVICE CURRENTLY PROCESSING THE REQUEST ?
YES
NO
3
CALLS THE USER-WRITTEN DEVICE STOP PROCEDURE
IS THE REQUEST DONE ?
YES
NO
4
REMOVES THE IORS FROM THE QUEUE
SENDS THE IORS TO THE RESPONSE MAILBOX
ANY MORE IORSs TO CANCEL ?
YES
NO
5
SURRENDERS ACCESS TO THE REGION
RETURNS TO THE I/O SYSTEM
1872

Figure A-4. Random Access Device Driver Cancel I/O Procedure

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 correct contents of the DS register, it passed the token of the data storage area as the contents of the task's DS register.

   - When the INIT$IO procedure created the data storage area, it included the correct contents of the DS register in one of the fields.

   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 field of the data storage area. Thus the task has the correct value in its DS register, and it has 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. For iRMX 86 systems, it makes an 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.

   For iRMX 88 systems, it makes an RQ$ELVL system call to enable the nucleus-provided default interrupt handler.

3. It begins an infinite loop by waiting for an interrupt of the specified level.

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 user-written 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 3.

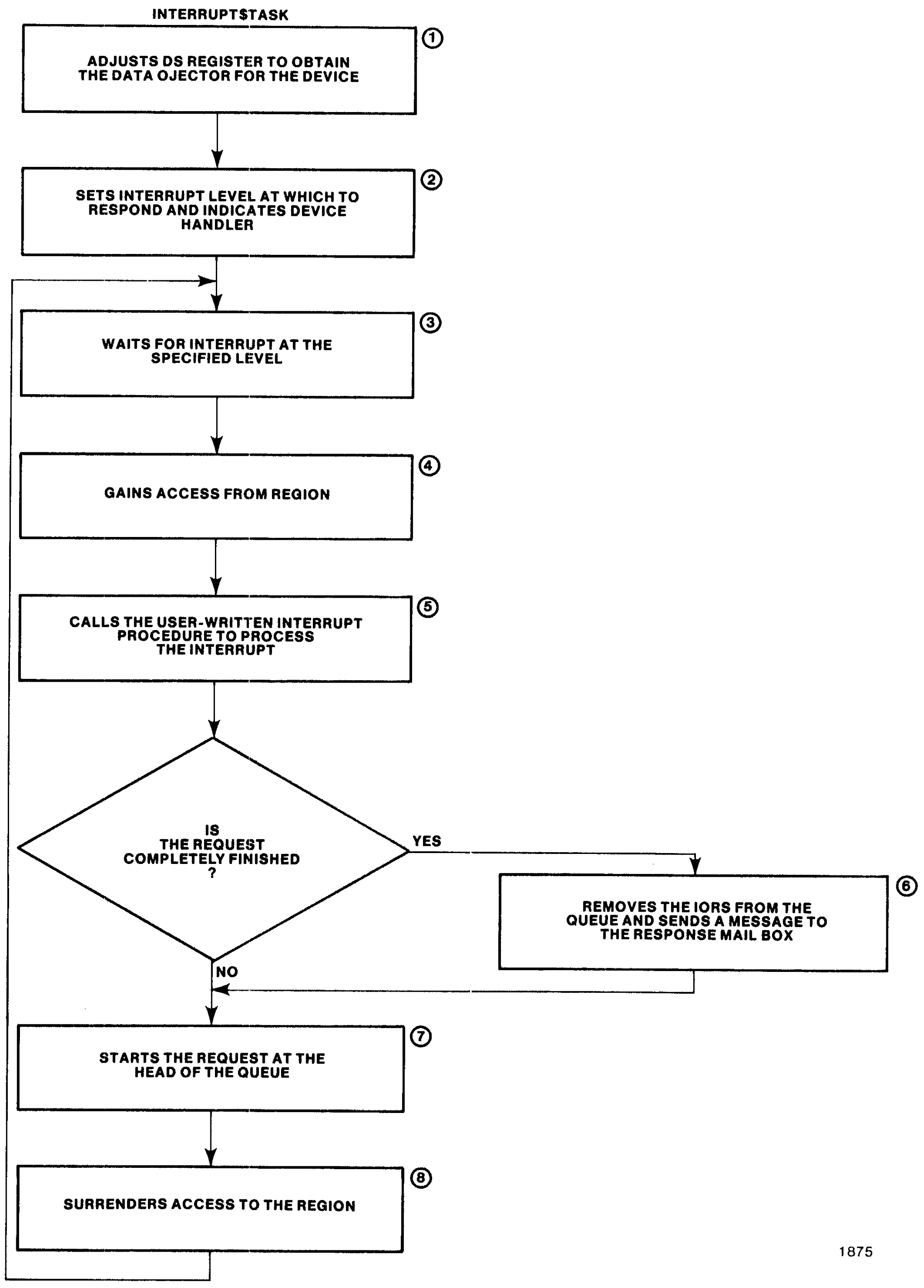

INTERRUPT$TASK
① ADJUSTS DS REGISTER TO OBTAIN THE DATA OJECTOR FOR THE DEVICE
② SETS INTERRUPT LEVEL AT WHICH TO RESPOND AND INDICATES DEVICE HANDLER
③ WAITS FOR INTERRUPT AT THE SPECIFIED LEVEL
④ GAINS ACCESS FROM REGION
⑤ CALLS THE USER-WRITTEN INTERRUPT PROCEDURE TO PROCESS THE INTERRUPT
IS THE REQUEST COMPLETELY FINISHED ?
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

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 user-written 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 EXAMPLES OF DEVICE DRIVERS

This appendix contains four examples of device drivers. The first example is a common driver which drives a line printer. The second is a random access driver, which drives a iSBC 206 disk controller. The third example is an 8274 terminal driver. (The contents of the INCLUDE files that these drivers use are listed in the last section of this appendix.)

Note that the names of the procedures in the examples are not device$start, device$interrupt, etc., as in the text of this manual. This is because the actual names are placed in the appropriate DUIBs during configuration.

Table B-1 lists the device driver example file names and the pages on which they appear.

Table B-1. Device Driver Examples

File	Description	Page
iprntr.p86	Driver for a line printer	B-2
i206ds.p86	Driver for an iSBC 206 disk controller	B-6
x8274.p86	8274 terminal driver	B-20
	INCLUDE files for above device drivers	B-39

```
PL/M-86 COMPILER    xprntr.p86


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE XPRNTR
OBJECT MODULE PLACED IN :F1:XPRNTR.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:XPRNTR.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE



                $title ('xprntr.p86')
                /*
                 *  xprntr.p86
                 *
                 *     This module implements centronix-type interface line printer
                 *     driver.  It is written as a 'common' device driver.  It is
                 *     assumed that the reader is familiar with the 8255 chip.
                 *
                 *
                 *  LANGUAGE DEPENDENCIES:
                 *     COMPACT ROM OPTIMIZE(3)
                 */

                /*
                 *            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.
                 *
                 */

   1            xprntr: DO;
                $include(:f1:xcomon.lit)
           =    $save nolist
                $include(:f1:xparam.lit)
           =    $save nolist
                $include(:f1:xnutyp.lit)
           =    $save nolist
                $include(:f1:xiors.lit)
           =    $save nolist
                $include(:f1:xduib.lit)
           =    $save nolist
                $include(:f1:xprntr.lit)
           =    $save nolist
                $include(:f1:x8255.lit)
           =    $save nolist
                $include(:f1:xprerr.lit)
           =    $save nolist
                $include(:f0:nsleep.ext)
           =    $SAVE NOLIST
                /*
                 *  literal declaration
                 */
  23    1       DECLARE
                   TAB$CHAR LITERALLY '09H',
                   SPACE    LITERALLY '20H';


                $subtitle('printer$start$interrupt')

                /*
                 *  printer$start/printer$interrupt
                 *       start/interrupt procedure for the line printer
                 *
                 *  CALLING SEQUENCE:
                 *       CALL printer$start$interrupt (iors$p, duib$p, ddata$p);
                 *
```

```
 *  INTERFACE VARIABLES:
 *       iors$p   -   I/O request/result segment pointer
 *       duib$p   -   pointer to the device-unit info. block
 *       ddata$p  -   pointer to the device(printer) data segment.
 *
 *  CALLS:  None
 *
 */

printer$start$interrupt: PROCEDURE (iors$p, duib$p, ddata$p)
                                                    PUBLIC REENTRANT;
   DECLARE
      (iors$p, duib$p, ddata$p)   POINTER;
   DECLARE
      iors     BASED  iors$p  IO$REQ$RES$SEG,
      duib     BASED  duib$p  DEV$UNIT$INFO$BLOCK;
   DECLARE
      dinfo$p  POINTER,
      dinfo    BASED  dinfo$p PRINTER$DEVICE$INFO;
   DECLARE
       buffer$p     POINTER,
      (char  BASED  buffer$p)(1) BYTE;

   dinfo$p = duib.device$info$p;

   /*
    *  test for spurious interrupts
    */
   IF iors$p = 0 THEN
   DO;
      /*
       * turn off the interrupt and return
       */
      OUTPUT(dinfo.Control$port) = INT$DISABLE;
      RETURN;
   END;

   DO CASE (iors.funct);

      /* read */
      DO;
         iors.status = E$IDDR;
         iors.done = TRUE;
      END;

      /* write */
      DO;
         /* get the buffer pointer */
         buffer$p = iors.buff$p;




         /* disable printer interrupt */
         OUTPUT(dinfo.Control$port) = INT$DISABLE;

         DO WHILE (iors.actual < iors.count);

            /*
             *  test for printer ready and not paper out. if not ready
             *  or paper out then wait forever.
             */
            DO WHILE (((INPUT(dinfo.C$port) AND PRINTER$READY) = 0) OR
                        ((INPUT(dinfo.C$port) AND PAPER$OUT) <> 0));
               /*  sleep for 100 nucleus clock intervals */
               CALL rq$sleep(100, @iors.status);
            END;
```

```
          /*
           *  convert TAB character to a SPACE character if the
           *  printer does not handle them
           */
          IF ((char(iors.actual) = TAB$CHAR) AND
                                     ((dinfo.tab$control) = FALSE))
             THEN char(iors.actual) = SPACE;
          /*
           * 1's complement the character and send it to the
           * printer.  Port-A is the data port
           */
          OUTPUT(dinfo.A$port) = NOT(char(iors.actual));
          /*
           * strobe the line printer
           * this is a way of telling the printer that there is
           * valid data on the bus
           */
          OUTPUT(dinfo.Control$port) = STROBE$ON;
          OUTPUT(dinfo.Control$port) = STROBE$OFF;
          /*
           *  increment the count of chars printed
           */
          iors.actual = iors.actual + 1;
          /*
           *  test whether printer acknowledgement bit is set
           */
          IF (INPUT(dinfo.C$port) AND CHAR$ACK AND CHAR$ACK$COMPLETE) = 0
             THEN
          DO;
             /*
              *  printer didn't acknowledge. Hopefully it has
              *  started printing. So enable the printer interrupt
              *  and return(printer will interrupt when it's done)
              */
             OUTPUT(dinfo.Control$port) = INT$ENABLE;
             RETURN;
          END;
       END;  /* end of DO WHILE statement */

       /*
        *  set iors.done to TRUE
        *  set iors.status to OK
        */
       iors.status = E$OK;
       iors.done = TRUE;
    END;

    /* seek */
    DO;
       iors.status = E$IDDR;
       iors.done = TRUE;
    END;

    /* special */
    DO;
       iors.status = E$IDDR;
       iors.done = TRUE;
    END;

    /* attach device */
    DO;
       /* initialize the 8255 */
       OUTPUT(dinfo.Control$port) = MODE$WORD;
       iors.status = E$OK;
       iors.done = TRUE;
    END;
```

```
                  /* detach device */
    3             DO;
    4                iors.status = E$OK;
    4                iors.done = TRUE;
    4             END;

                  /* open */
    3             DO;
    4                iors.status = E$OK;
    4                iors.done = TRUE;
    4             END;

                  /* close */
    3             DO;
    4                iors.status = E$OK;
    4                iors.done = TRUE;
    4             END;

    3          END;  /* end of DO CASE statement */

    2       END printer$start$interrupt;


            /*
             *  printer$stop
             *       stop procedure for the line printer
             *
             *  CALLING SEQUENCE:
             *       CALL printer$stop (iors$p, duib$p, ddata$p);
             *
             *  INTERFACE VARIABLES:
             *       iors$p   -   I/O request/result segment pointer
             *       duib$p   -   pointer to the device-unit info. block
             *       ddata$p  -   pointer to the device(printer) data segment.
             *
             *  CALLS:  None
             *
             */

    1       printer$stop: PROCEDURE (iors$p, duib$p, ddata$p) PUBLIC REENTRANT;
    2          DECLARE
                  (iors$p, duib$p, ddata$p)   POINTER;
    2          DECLARE
                  iors     BASED  iors$p  IO$REQ$RES$SEG,
                  duib     BASED  duib$p  DEV$UNIT$INFO$BLOCK;
    2          DECLARE
                  dinfo$p  POINTER,
                  dinfo    BASED  dinfo$p PRINTER$DEVICE$INFO;

               /*
                *  turn off the printer interrupt
                *  set iors.done to TRUE
                *  set iors.status to E$OK
                */

    2          dinfo$p = duib.device$info$p;

    2          OUTPUT(dinfo.Control$port) = INT$DISABLE;
    2          iors.status = E$OK;
    2          iors.done = TRUE;

    2       END printer$stop;

    1       END xprntr;
```

```
PL/M-86 COMPILER    x206ds.p86
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206DS
OBJECT MODULE PLACED IN :F1:X206DS.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X206DS.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


              $title('x206ds.p86')
              $subtitle('Module Header')

              /*
               *  x206ds.p86
               *
               *  iSBC 206 device
               *
               * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
               */

   1          x206ds: DO;


              /*
               *             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.
               *
               */

              $include(:f1:xcomon.lit)
          =   $save nolist
              $include(:f1:xnutyp.lit)
          =   $save nolist
              $include(:f1:xparam.lit)
          =   $save nolist
              $include(:f1:xiotyp.lit)
          =   $save nolist
              $include(:f1:xiors.lit)
          =   $save nolist
              $include(:f1:xduib.lit)
          =   $save nolist
              $include(:f1:xdrinf.lit)
          =   $save nolist
              $include(:f1:x206in.lit)
          =   $save nolist
              $include(:f1:x206dv.lit)
          =   $save nolist
              $include(:f1:xexcep.lit)
          =   $save nolist
              $include(:f1:xioexc.lit)
          =   $save nolist
              $include(:f1:xradsf.lit)
          =   $save nolist

              $include(:f1:x206dp.ext)
          =   $save nolist
              $include(:f1:x206dc.ext)
          =   $save nolist
              $include(:f1:x206fm.ext)
          =   $save nolist
              $include(:f1:xnotif.ext)
          =   $save nolist
```

```
$subtitle('Local Data')

    /*
     *  need$reset array used to determine if device needs to be
     *  reset after an error. Indexed by status.
     */

    DECLARE
        need$reset(24)      BYTE DATA(
            FALSE,                      /* Successful completion */
            TRUE,                       /* ID field miscompare */
            FALSE,                      /* Data field CRC error */
            FALSE,                      /* special for incorrect result$type */
            TRUE,                       /* Seek error */
            FALSE,
            FALSE,
            FALSE,
            FALSE,                      /* Illegal Record Address */
            FALSE,
            FALSE,                      /* ID Field CRC error */
            TRUE,                       /* Protocol error */
            TRUE,                       /* Illegal Cylinder Address */
            FALSE,
            FALSE,                      /* Record not found */
            FALSE,                      /* Data Mark Missing */
            FALSE,                      /* Format Error */
            FALSE,                      /* Write Protected */
            FALSE,
            TRUE,                       /* Write Error */
            FALSE,
            FALSE,
            FALSE,
            FALSE);                     /* Drive Not Ready */

    /*
     *  unit$status is used to set unit status field in iors.
     *  Indexed by status.
     */

    DECLARE
        unit$status(24)     BYTE DATA(
            IO$UNCLASS,                 /* Successful completion */
            IO$SOFT,                    /* ID field miscompare */
            IO$SOFT,                    /* Data field CRC error */
            IO$HARD,                    /* special for incorrect result$type */
            IO$SOFT,                    /* Seek error */
            IO$UNCLASS,
            IO$UNCLASS,
            IO$UNCLASS,
            IO$HARD,                    /* Illegal Record Address */
            IO$UNCLASS,
            IO$SOFT,                    /* ID Field CRC error */
            IO$SOFT,                    /* Protocol error */
            IO$HARD,                    /* Illegal Cylinder Address */
            IO$UNCLASS,
            IO$SOFT,                    /* Record not found */
            IO$SOFT,                    /* Data Mark Missing */
            IO$SOFT,                    /* Format Error */
            IO$WRPROT,                  /* Write Protected */
            IO$UNCLASS,
            IO$SOFT,                    /* Write Error */
            IO$UNCLASS,
            IO$UNCLASS,
            IO$UNCLASS,
            IO$OPRINT);                 /* Drive Not Ready */

    /*
     *  drive$ready is used to find the drive ready bit
     *  in the drive status.
     */
```

```
        DECLARE
            drive$ready(4)  BYTE DATA(020H,040H,010H,020H);

    $subtitle('i206$start')

        /*
         * i206$start
         *      start procedure for the iSBC 206
         *
         * CALLING SEQUENCE:
         *      CALL i206$start(iors$p, duib$p, ddata$p);
         *
         * INTERFACE VARIABLES:
         *      iors$p      - I/O Request/Result segment pointer
         *      duib$p      - pointer to Device-Unit Information Block
         *      ddata$p     - device data segment pointer.
         *
         * CALLS:
         *      io$206
         *      format$206
         *      send$206$iopb
         *
         */

        i206$start: PROCEDURE(iors$p, duib$p, ddata$p) PUBLIC REENTRANT;
            DECLARE
                iors$p      POINTER,
                duib$p      POINTER,
                ddata$p     POINTER;
            DECLARE
                iors        BASED iors$p IO$REQ$RES$SEG,
                duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                dinfo$p     POINTER,
                dinfo       BASED dinfo$p I206$DEVICE$INFO,
                uinfo$p     POINTER,
                uinfo       BASED uinfo$p I206$UNIT$INFO,
                ddata       BASED ddata$p IO$PARM$BLOCK$206,
                base        WORD,
                dummy       BYTE;

            dinfo$p = duib.device$info$p;
            base = dinfo.base;
            uinfo$p = duib.unit$info$p;

            IF (ddata.restore) THEN
                RETURN;

        do$case$funct:
            DO CASE iors.funct;

                /*
                 * in the following calls the @ddata is literally
                 * iopb$p (i.e., the pointer to the iopb).
                 */

            case$read:
                DO;
                    CALL io$206(base, iors$p, duib$p, @ddata);
                END case$read;

            case$write:
                DO;
                    CALL io$206(base, iors$p, duib$p, @ddata);
                END case$write;

            case$seek:
                DO;
                    CALL io$206(base, iors$p, duib$p, @ddata);
                END case$seek;
    case$spec$funct:
        DO;
            IF iors.sub$funct = FS$FORMAT$TRACK THEN
                CALL format$206(base, iors$p, duib$p, @ddata);
            ELSE
                DO;
                    iors.status = E$IDDR;
                    iors.actual = 0;
                    iors.done = TRUE;
                END;
        END case$spec$funct;

    case$attach$device:
        DO;
            dummy = (duib.dev$gran = 512);
            IF ((input(sub$system$port) OR 073H) <> 0FBH) OR
                (((input(disk$config$port) AND SHL(010H,SHR(duib.unit,2))) <> 0) <> dummy) THEN
                DO;
                    iors.status = E$IO;
                    iors.unit$status = IO$OPRINT;
                    iors.actual = 0;
                    iors.done = TRUE;
                    RETURN;
                END;
            ddata.inter = inter$on$mask;
            ddata.instr = restore$op;
            IF NOT send$206$iopb(base, @ddata) THEN
                /*
                 * the board would not accept the iopb
                 * so...
                 */
                DO;
                    iors.status = E$IO;
                    iors.unit$status = IO$SOFT OR SHL(input(result$byte$port), 8);
                    iors.actual = 0;
                    iors.done = TRUE;
                END;
        END case$attach$device;

    case$detach$device:
        DO;
            iors.status = E$OK;
            iors.done = TRUE;
        END case$detach$device;

    case$open:

        DO;
            iors.status = E$OK;
            iors.done = TRUE;
        END case$open;

    case$close:
        DO;
            iors.status = E$OK;
            iors.done = TRUE;
        END case$close;

    END do$case$funct;

END i206$start;
```

```
$subtitle('i206$interrupt')

    /*
     * i206$interrupt
     *      interrupt procedure for the iSBC 206
     *
     * CALLING SEQUENCE:
     *      CALL i206$interrupt(iors$p, duib$p, ddata$p);
     *
     * INTERFACE VARIABLES:
     *      iors$p      - I/O Request/Result segment pointer
     *      duib$p      - pointer to Device-Unit Information Block
     *      ddata$p     - device data segment pointer.
     *
     * CALLS:
     *      i206$start
     *      send$206$iopb
     *      rq$send$message
     *
     */

    i206$interrupt: PROCEDURE(iors$p, duib$p, ddata$p) PUBLIC REENTRANT;
        DECLARE
            iors$p          POINTER,
            duib$p          POINTER,
            ddata$p         POINTER;
        DECLARE
            iors        BASED iors$p IO$REQ$RES$SEG,
            duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
            dinfo$p     POINTER,
            dinfo       BASED dinfo$p I206$DEVICE$INFO,
            ddata       BASED ddata$p IO$PARM$BLOCK$206,
            temp        BYTE,
            base        WORD,
            spindle     WORD,
            status      WORD;

        dinfo$p = duib.device$info$p;
        base = dinfo.base;
        spindle = shr(duib.unit,.2);                         /* 4 units/spindle */

        IF (input(result$type$port) AND 3) = 0 THEN
        done$int:
            DO;
                status = input(result$byte$port);

                IF ddata.restore THEN
                did$restore:
                    DO;
                        ddata.restore = FALSE;
                        ddata.status(spindle) = status;
                        IF iors$p <> 0 THEN
                        restart:
                            DO;
                                CALL i206$start(iors$p, ddata$p, duib$p);
                            END restart;
                        RETURN;
                    END did$restore;

                ddata.status(spindle) = status;

                IF iors$p <> 0 THEN
                valid$iors:
                    DO;
                        IF status <> 0 THEN
                        bad$status:
                            DO;
```

```
5                                iors.status = E$IO;
5                                IF (status <= 010H) THEN
5                                    temp = status;
5                                ELSE
                                     temp = shr(status, 4) + 00FH;
5                                iors.unit$status = unit$status(temp) OR SHL(status,8);
5                                iors.actual = 0;
5                                iors.done = TRUE;
5                                IF need$reset(ddata.status(iors.unit / 4)) THEN
5                                recalibrate:
                                     DO;
                                         /*
                                          * Note: must index drive select
                                          * bits from iors.unit.
                                          */
6                                        ddata.inter = inter$on$mask;
6                                        ddata.instr = restore$op;
6                                        ddata.restore = send$206$iopb(dinfo.base, @ddata);
6                                    END recalibrate;

5                            END bad$status;
4                        ELSE ok$status:
                             DO;
5                                iors.actual = iors.count;
5                                iors.done = TRUE;
5                            END ok$status;
4                  END valid$iors;
3              END done$int;
2          ELSE status$int:
               DO;
3                  temp = input(inter$stat$port);
3                  DO spindle=0 TO 3;
4                      IF (temp AND SHL(1, spindle)) <> 0 THEN
4                          GOTO found$spindle;
4                  END;
3      found$spindle:
                   spindle = SHL(spindle,2);
3                  DO temp=spindle TO spindle+3;
4                      IF ((input(result$byte$port) AND drive$ready(spindle)) = 0) THEN
4                          CALL notify(temp, @ddata);
4                  END;
3              END status$int;

2        END i206$interrupt;
       $subtitle('i206$init')

         /*
          * i206$init
          *      init procedure for the iSBC 206
          *
          * CALLING SEQUENCE:
          *      CALL i206$init(duib$p, ddata$p, status$p);
          *
          * INTERFACE VARIABLES:
          *      duib$p      - pointer to Device-Unit Information Block
          *      ddata$p     - device data segment pointer.
          *      status$p    - pointer to WORD indicating status of operation
          *
          * CALLS:
          *      <none>
          *
          */

1        i206$init: PROCEDURE(duib$p, ddata$p, status$p) PUBLIC REENTRANT;
2            DECLARE
                 duib$p          POINTER,
                 ddata$p         POINTER,
                 status$p        POINTER;
```

```
 169   2             DECLARE
                         duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                         dinfo$p     POINTER,
                         dinfo       BASED dinfo$p I206$DEVICE$INFO,
                         ddata       BASED ddata$p IO$PARM$BLOCK$206,
                         status      BASED status$p WORD;
 170   2             DECLARE
                         i           WORD;

 171   2             dinfo$p = duib.device$info$p;

                     /*
                      * Reset 206;  not there or not hard disk ==> Oops!
                      */

 172   2             output(reset$port) = 0;
 173   2             status = E$OK;

 174   2             ddata.restore = FALSE;

 175   2         END i206$init;

 176   1     END x206ds;



MODULE INFORMATION:

     CODE AREA SIZE     = 0300H    768D
     CONSTANT AREA SIZE = 0034H     52D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0046H     70D
     1037 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     18KB MEMORY USED   (18%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION


PL/M-86 COMPILER    x206io.p86: iSBC 206 I/O Module
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206IO
OBJECT MODULE PLACED IN :F1:X206IO.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X206IO.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


             $title('x206io.p86: iSBC 206 I/O Module')
             $subtitle('Module Header')
   1         x206io: DO;


             /*
              *            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.
              *
              */
```

```
                 /*
                  *  This module modifies the 206 parameter block and passes the
                  *      address of it to the iSBC 206.
                  *
                  * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
                  */

                 $include(:f1:xcomon.lit)
             =   $save nolist
                 $include(:f1:xnutyp.lit)
             =   $save nolist
                 $include(:f1:xiotyp.lit)
             =   $save nolist
                 $include(:f1:xparam.lit)
             =   $save nolist
                 $include(:f1:x206dv.lit)
             =   $save nolist
                 $include(:f1:x206in.lit)
             =   $save nolist
                 $include(:f1:xiors.lit)
             =   $save nolist
                 $include(:f1:xduib.lit)
             =   $save nolist
                 $include(:f1:xtrsec.lit)
             =   $save nolist
                 $include(:f1:xexcep.lit)
             =   $save nolist
                 $include(:f1:xioexc.lit)
             =   $save nolist
                 $include(:f1:x206dc.ext)
             =   $save nolist

                 /*
                  * this module also does seeks
                  */

 32   1              DECLARE
                         i206$op$codes (*)   BYTE DATA(
                             READ$OP,
                             WRITE$OP,
                             SEEK$OP
                         );

                 $subtitle('io$206: iSBC 206 I/O Module')

                     /*
                      * io$206
                      *      I/O module (read/write/seek)
                      *
                      * CALLING SEQUENCE:
                      *      CALL io$206 (base, iors$p, duib$p, iopb$p);
                      *
                      * INTERFACE VARIABLES:
                      *      base        - base address of the board.
                      *      iors$p      - I/O Request/Result segment pointer
                      *      duib$p      - pointer to Device-Unit Information Block
                      *      iopb$p      - pointer to I/O parameter block.
                      *
                      * CALLS:
                      *      send$206$iopb(base, @iopb)
                      */
 33   1              io$206: PROCEDURE (base, iors$p, duib$p, iopb$p) REENTRANT PUBLIC;
 34   2                  DECLARE
                             base        WORD,
                             iors$p      POINTER,
                             duib$p      POINTER,
                             iopb$p      POINTER;
```

```
                DECLARE
                    iors        BASED iors$p IO$REQ$RES$SEG,
                    ts          DWORD,
                    ts$o        TRACK$SECTOR$STRUCT AT(@ts),
                    duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                    iopb        BASED iopb$p IO$PARM$BLOCK$206,
                    platter     BYTE,
                    spindle     BYTE,
                    surface     BYTE;

                ts = iors.dev$loc;

                spindle = shr(iors.unit, 2);                    /* 4 units/spindle */
                platter = iors.unit AND 003H;                   /* (as above ) */
                surface = ts$o.track AND 00001H;                /* select surface */

                iopb.inter = INTER$ON$MASK;
                iopb.cyl$add = shr(ts$o.track, 1);              /* track/2 = cylinder */
                iopb.instr = i206$op$codes(iors.funct)         OR
                             shl(spindle, 4) OR
                             shl(platter, 6) OR
                             shl(surface, 3);

                /*
                 * note: the controller only supports 512 or 128 byte sectors
                 * so no checking is done.
                 */

                iopb.r$count = iors.count / duib.dev$gran;      /* divide by sectors size */
                iopb.rec$add = (ts$o.sector + 1) OR
                                 shr(ts$o.track AND 0200H, 2);  /* (cyl AND 0100H) / 2 */
                iopb.buff$p = iors.buff$p;

                IF NOT send$206$iopb(base, @iopb) THEN

                    /*
                     * the board did not accept the iopb so...
                     */
                    DO;
                        iors.status = IO$SOFT;
                        iors.actual = 0;
                        iors.done = TRUE;
                    END;

            END io$206;

        END x206io;


MODULE INFORMATION:

     CODE AREA SIZE     = 00D5H    213D
     CONSTANT AREA SIZE = 0003H      3D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0022H     34D
     605 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     12KB MEMORY USED   (12%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER    x206dc: iSBC 206 parameter handler
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206DC
OBJECT MODULE PLACED IN :F1:X206DC.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X206DC.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


             $title('x206dc: iSBC 206 parameter handler')
             $subtitle('Module Header')
             x206dc: DO;


             /*
              *             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.
              *
              */

             /*
              *  This module contains the commands for the 206 controller.
              *
              * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
              */

             $include(:f1:xcomon.lit)
         =   $save nolist
             $include(:f1:xnutyp.lit)
         =   $save nolist
             $include(:f1:x206dv.lit)
         =   $save nolist

             $subtitle('Send 206 I/O Parameter Block')

                 /*
                  * send$206$iopb
                  *      send the iSBC 206 the address of the parameter block
                  *
                  * CALLING SEQUENCE:
                  *      CALL send$206$iopb (base, iopb$p);
                  *
                  * INTERFACE VARIABLES:
                  *      base        - base address of board.
                  *      iopb$p      - I/O parameter block pointer
                  *
                  * CALLS:
                  *      (none)
                  */
                 send$206$iopb: PROCEDURE (base, iopb$p) BOOLEAN REENTRANT PUBLIC;
                     DECLARE
                         base        WORD,
                         iopb$p      POINTER;
                     DECLARE
                         iopb$p$o    P$OVERLAY AT(@iopb$p),
                         iopb        BASED iopb$p IO$PARM$BLOCK$206,
                         drive       BYTE;

                     drive = shr(iopb.instr AND 030H, 4);
                     drive = shl(01H,drive);

                     IF (input(controller$stat)) = (COMMAND$BUSY OR drive) THEN
                         RETURN(FALSE);

                     output (lo$off$port) = low (iopb$p$o.offset);
```

```
 17   2              IF (input(controller$stat) AND COMMAND$BUSY) <> 0 THEN
 18   2                 RETURN(FALSE);

                     /*
                      * made it to here so output rest of iopb address
                      */

 19   2              output (lo$seg$port) = low (iopb$p$o.base);
 20   2              output (hi$seg$port) = high (iopb$p$o.base);
 21   2              output (hi$off$port) = high (iopb$p$o.offset);

 22   2              RETURN(TRUE);

 23   2           END send$206$iopb;

 24   1        END x206dc;


MODULE INFORMATION:

     CODE AREA SIZE     = 0060H     96D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 000CH     12D
     208 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     5KB MEMORY USED   (5%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION



PL/M-86 COMPILER    x206fm.p86
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206FM
OBJECT MODULE PLACED IN :F1:X206FM.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X206FM.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


               $title('x206fm.p86')
               $subtitle('Module Header')

               /*
                *  x206fm.p86
                *
                *  iSBC 206 device
                *      formats one track on hard disk
                *
                * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
                */

   1           x206fm: DO;
```

```
     /*
      *             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.
      *
      */

     /*
      *  This module builds the 206 parameter block and passes the
      *      address of it to the iSBC 206.
      *
      *      note: this format procedure deduces max$sectors from the
      *      DEV$UNIT$INFO$BLOCK (duib.dev$gran).  it does NOT check to see if
      *      the operator has set the switches on the controller correctly.
      *      sectors may be 128 or 512 bytes.
      *
      */

     $include(:f1:xcomon.lit)
=    $save nolist
     $include(:f1:xnutyp.lit)
=    $save nolist
     $include(:f1:xiotyp.lit)
=    $save nolist
     $include(:f1:xparam.lit)
=    $save nolist
     $include(:f1:x206dv.lit)
=    $save nolist
     $include(:f1:x206in.lit)
=    $save nolist
     $include(:f1:xradsf.lit)
=    $save nolist
     $include(:f1:xiors.lit)
=    $save nolist
     $include(:f1:xduib.lit)
=    $save nolist
     $include(:f1:xtrsec.lit)
=    $save nolist
     $include(:f1:xexcep.lit)
=    $save nolist
     $include(:f1:xioexc.lit)
=    $save nolist

     $include(:f1:x206dc.ext)
=    $save nolist

     $subtitle('format$206: Format track procedure')

         /*
          * format$206
          *      format a track on the 206
          *
          * CALLING SEQUENCE:
          *      CALL format$206 (base, iors$p, duib$p, iopb$p);
          *
          * INTERFACE VARIABLES:
          *      base        - base address of board.
          *      iors$p      - I/O Request/Result segment pointer
          *      duib$p      - pointer to Device-Unit Information Block
          *      iopb$p      - I/O parameter block pointer.
          *
          * CALLS:
          *      build$206$fmt$table
          *      send$206$iopb
          */
```

```
format$206: PROCEDURE (base, iors$p, duib$p, iopb$p) REENTRANT PUBLIC;
    DECLARE
        base            WORD,
        iors$p          POINTER,
        duib$p          POINTER,
        iopb$p          POINTER;
    DECLARE
        iors        BASED iors$p IO$REQ$RES$SEG,
        format$info$p   POINTER,
        format$info BASED format$info$p FORMAT$INFO$STRUCT,
        duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
        iopb        BASED iopb$p IO$PARM$BLOCK$206,
        platter             BYTE,
        spindle             BYTE,
        surface             BYTE,
        max$sectors         BYTE;

    format$info$p = iors.aux$p;
    IF format$info.track$num > i206$TRACK$MAX THEN
        DO;
            iors.status = E$SPACE;
            iors.actual = 0;
            iors.done = TRUE;
            RETURN;
        END;

    spindle = shr(iors.unit, 2);                     /* 4 units/spindle */
    platter = iors.unit AND 003H;                    /* (as above ) */
    surface = format$info.track$num AND 00001H;      /* select surface */

    iopb.inter = INTER$ON$MASK OR FORMAT$TRACK$ON;
    iopb.cyl$add = shr(format$info.track$num, 1);   /* track/2 = cylinder */
    iopb.rec$add = shr(format$info.track$num AND 0200H, 2); /* set if over 256 cylinders */
    iopb.instr = format$op          OR
                 shl(spindle, 4) OR
                 shl(platter, 6) OR
                 shl(surface, 3);

    iopb.buff$p = @iopb.format$table;

    IF duib.dev$gran = 128 THEN
        max$sectors = 36;
    ELSE
        /*
         * if not 128 then MUST be 512 byte sectors
         */
        max$sectors = 12;

    CALL build$206$fmt$table(@iopb.format$table,
                             format$info.track$num,
                             format$info.track$interleave,
                             format$info.track$skew,
                             format$info.fill$char,
                             max$sectors);

    IF NOT send$206$iopb(base, @iopb) THEN
        /*
         * the board did not accept the iopb so...
         */
        DO;
            iors.status = IO$SOFT;
            iors.actual = 0;
            iors.done = TRUE;
        END;

END format$206;
```

```
          /*
           * build$206$fmt$table
           *      fill out format table
           *
           * CALLING SEQUENCE:
           *      CALL build$206$fmt$table(buf$p, track, int$fact, skew, fill$char, max$sectors);
           *
           * INTERFACE VARIABLES:
           *      buf$p       - address of format table.
           *      track       - track to be formatted.
           *      int$fact    - interleave factor.
           *      skew        - squew from physical  sector one.
           *      fill$char   - used to fill sectors.
           *      max$sectors - maximum number of sectors
           *
           * CALLS:
           *      <none>
           *
           * No error checking on skew, int$fact parameters; if nonsense, the algorithm
           * completes & formats the track in a strange manner.
           */

          build$206$fmt$table: PROCEDURE(buf$p, track, int$fact, skew, fill$char,max$sectors) REENTRANT;
              DECLARE
                  buf$p       POINTER,
                  track       WORD,
                  int$fact    BYTE,
                  skew        BYTE,
                  fill$char   BYTE,
                  max$sectors BYTE;
              DECLARE
                  s           BYTE,
                  i           BYTE;
              DECLARE
                  fmt$tab BASED buf$p (36) STRUCTURE(
                      record$address  BYTE,
                      fill$char       BYTE);

              DO i = 0 TO (max$sectors - 1);
                  fmt$tab(i).record$address = 0FFH;
                  fmt$tab(i).fill$char = fill$char;
              END;

              s = skew MOD max$sectors;

              DO i = 1 TO max$sectors;

                  DO WHILE fmt$tab(s).record$address <> 0FFH;
                      s = (s + 1) MOD max$sectors;
                  END;

                  fmt$tab(s).record$address = i;
                  s = (s + int$fact) MOD max$sectors;
              END;

          END build$206$fmt$table;

      END x206fm;


              MODULE INFORMATION:

                   CODE AREA SIZE     = 0192H    402D
                   CONSTANT AREA SIZE = 0000H      0D
                   VARIABLE AREA SIZE = 0000H      0D
                   MAXIMUM STACK SIZE = 0028H     40D
                   743 LINES READ
                   0 PROGRAM WARNINGS
                   0 PROGRAM ERRORS

              DICTIONARY SUMMARY:

                   96KB MEMORY AVAILABLE
                   13KB MEMORY USED   (13%)
                   0KB DISK SPACE USED

              END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER    x8274: 8274 terminal device driver
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X8274
OBJECT MODULE PLACED IN :F1:X8274.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X8274.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


             $title('x8274: 8274 terminal device driver')
             $subtitle('Module Header')

             /*
              * TITLE:      x8274
              *
              * DATE:       27 FEB 84
              *
              * ABSTRACT:   This module is the interface between the iRMX 86
              *             Terminal Support, and the 8274 MPSC.  It is a
              *             rewritten version of a module of the same name dated
              *             20 Jan 83.  The rewritting was necessary to correct
              *             initialization timing problems and to add support for
              *             various timer devices, i.e. 8253-4, 80130, and 80186-8.
              *
              * LANGUAGE DEPENDENCIES:   PLM86 COMPACT ROM
              */

             /*
              *              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.
              *
              */

  1           x8274: DO;

             $include(:f1:xcomon.lit)
         =   $save nolist
             $include(:f1:xnutyp.lit)
         =   $save nolist
             $include(:f1:xiotyp.lit)
         =   $save nolist
             $include(:f1:xexcep.lit)
         =   $save nolist

             $include(:f1:xtssow.ext)
         =   $save nolist
             $include(:f1:xtstim.ext)
         =   /*
         =    * External Declaration
         =    * for timer support procedure.
         =    */
         =   $SAVE NOLIST
             $include(:f1:xdelay.ext)
         =   $SAVE NOLIST

             $subtitle('Data structures and literals')

             /*
              *  8274 register values
              */
```

```
20   1       DECLARE
                 WR0                        LITERALLY '00H',
                 WR1                        LITERALLY '01H',
                 WR2                        LITERALLY '02H',
                 WR3                        LITERALLY '03H',
                 WR4                        LITERALLY '04H',
                 WR5                        LITERALLY '05H',
                 WR6                        LITERALLY '06H',
                 WR7                        LITERALLY '07H',
                 RR0                        LITERALLY '00H',
                 RR1                        LITERALLY '01H',
                 RR2                        LITERALLY '02H';

             /*
              *  8274 command values
              */

21   1       DECLARE
                 NULL_CMD                   LITERALLY '00H',
                 NULL_VECTOR                LITERALLY '00H',
                 RESET_EXT_INT              LITERALLY '10H',
                 CHANNEL_RESET              LITERALLY '18H',
                 ENABLE_INT_NEXT_RX         LITERALLY '20H',
                 RESET_TX_INT               LITERALLY '28H',
                 ERROR_RESET                LITERALLY '30H',
                 END_OF_INT                 LITERALLY '38H';

             /*
              *  8274 write register commands.
              */

22   1       DECLARE
                 WR1_INIT                   LITERALLY '016H',    /* int on all Rx chars and
                                                                  * special conditions.
                                                                  * Parity affects vector,
                                                                  * variable vector,
                                                                  * Tx int enable, No
                                                                  * external int.
                                                                  */
                 WR1_NO_RX_INT              LITERALLY '006H',    /* disable Rx interrupts */
                 WR1_NO_INT                 LITERALLY '004H',    /* Disable Rx and Tx
                                                                  * interrupts
                                                                  */
                 WR2_INIT                   LITERALLY '004H',    /* non vectored int */
                 WR3_INIT                   LITERALLY '0C1H',    /* Rx 8 bits/char,
                                                                  * Rx enable
                                                                  */
                 WR3_RX_DISABLE             LITERALLY '0C0H',
                 WR4_INIT                   LITERALLY '044H',    /* 16X clock, 8 bit data,
                                                                  * 1 stop bit, no parity
                                                                  */
                 WR5_TX_ENABLE              LITERALLY '0EAH',    /* Tx 8 bits data,
                                                                  * Tx enable, RTS enable
                                                                  */
                 WR5_TX_DISABLE             LITERALLY '0E2H',
                 WR5_DTR_ON                 LITERALLY '0EAH',
                 WR5_DTR_OFF                LITERALLY '06AH';

             /*
              *  status register bit masks
              */
23   1       DECLARE
                 VECTOR_MASK            LITERALLY '0E0H',
                 TEST_VECTOR            LITERALLY '0A5H',
                 INT_PENDING            LITERALLY '002H',
                 NO_INT_VECT            LITERALLY '01CH',
                 RX_CHAR_RDY            LITERALLY '001H',
                 TX_BUFFER_EMPTY        LITERALLY '004H',
                 i8274$INPUT$ERROR      LITERALLY '070H';
              /*
               *  Flags values
               */

24    1       DECLARE
                  EVEN$MODE                 LITERALLY '003H',
                  ODD$MODE                  LITERALLY '001H',
                  NO$PARITY$MODE            LITERALLY '000H',
                  IN$PARITY$MASK            LITERALLY '030H',
                  OUT$PARITY$MASK           LITERALLY '1C0H',
                  STRIP$INPUT$PARITY$MODE   LITERALLY '000H',
                  PASS$INPUT$PARITY$MODE    LITERALLY '010H',
                  EVEN$INPUT$PARITY$MODE    LITERALLY '020H',
                  ODD$INPUT$PARITY$MODE     LITERALLY '030H',
                  SPACE$OUTPUT$PARITY$MODE     LITERALLY '000H',
                  MARK$OUTPUT$PARITY$MODE   LITERALLY '040H',
                  EVEN$OUTPUT$PARITY$MODE   LITERALLY '080H',
                  ODD$OUTPUT$PARITY$MODE    LITERALLY '0C0H',
                  PASS$OUTPUT$PARITY$MODE   LITERALLY '100H',
                  OUT$PAR$CHECK             LITERALLY '080H';

              /*
               *  Baud rate values
               */

25    1       DECLARE
                  HARDWARE$BAUD$SELECT      LITERALLY '0',
                  AUTO$BAUD$SELECT          LITERALLY '3',
                  OUT$BAUD$SAME             LITERALLY '1';

              /*
               *  interface to terminal support
               */

26    1       DECLARE
                  MORE$INTERRUPT            LITERALLY '08H',
                  NO$INTERRUPT              LITERALLY '00H',
                  DELAY$INTERRUPT           LITERALLY '05H + MORE$INTERRUPT',
                  INPUT$INTERRUPT           LITERALLY '01H + MORE$INTERRUPT',
                  OUTPUT$INTERRUPT          LITERALLY '02H + MORE$INTERRUPT',
                  RING$INTERRUPT            LITERALLY '03H + MORE$INTERRUPT',
                  CARRIER$INTERRUPT         LITERALLY '04H + MORE$INTERRUPT';


              /*
               * Controller Data Structure
               */

27    1       DECLARE
                  TS$CDATA     LITERALLY     'STRUCTURE(
                                                        TS$CDATA1,
                                                        TS$CDATA2)';

28    1       DECLARE
                  TS$CDATA1    LITERALLY    'ics$data$segment        SEGMENT,
                                             status                  WORD,
                                             interrupt$type          BYTE,
                                             interrupting$unit       BYTE,
                                             dinfo$p                 POINTER,
                                             driver$cdata$p          POINTER',
                  TS$CDATA2    LITERALLY    'reserved(34)            BYTE,
                                             udata(1)                BYTE';
```

```
             /*
              * Unit Data Structure
              */

29   1       DECLARE
                 TS$UDATA    LITERALLY   'STRUCTURE(
                                                     TS$UDATA1,
                                                     TS$UDATA2,
                                                     TS$UDATA3)';

30   1       DECLARE
                 TS$UDATA1   LITERALLY   'uinfo$p                  POINTER,
                                          term$flags               WORD,
                                          in$rate                  WORD,
                                          out$rate                 WORD,
                                          scroll$number            WORD,
                                          translation(87)          BYTE',

                 TS$UDATA2   LITERALLY   'input$control$table(33)     BYTE,
                                          unit$number              BYTE',

                 TS$UDATA3   LITERALLY   'fill(891)                BYTE';

             /*
              *  8274 Device information Structure
              */

31   1       DECLARE
                 i8274$CONTROLLER$INFO   LITERALLY   'STRUCTURE(
                                                                 i8274$INFO$1,
                                                                 i8274$INFO$2,
                                                                 i8274$INFO$3,
                                                                 i8274$INFO$4,
                                                                 i8274$INFO$5,
                                                                 i8274$INFO$6,
                                                                 i8274$INFO$7)';

32   1       DECLARE
                 i8274$INFO$1    LITERALLY   'filler(12)         WORD',
                 i8274$INFO$2    LITERALLY   'ch_a_data_port     WORD,
                                              ch_a_status_port   WORD,
                                              ch_b_data_port     WORD,
                                              ch_b_status_port   WORD',
                 i8274$INFO$3    LITERALLY   'ch_a_in_rate_port  WORD,
                                              ch_a_in_rate_cmd_port  WORD,
                                              ch_a_in_rate_counter   BYTE,
                                              ch_a_in_rate_freq  DWORD',
                 i8274$INFO$4    LITERALLY   'ch_a_out_rate_port WORD,
                                              ch_a_out_rate_cmd_port WORD,
                                              ch_a_out_rate_counter  BYTE,
                                              ch_a_out_rate_freq DWORD',
                 i8274$INFO$5    LITERALLY   'ch_b_in_rate_port  WORD,
                                              ch_b_in_rate_cmd_port  WORD,
                                              ch_b_in_rate_counter   BYTE,
                                              ch_b_in_rate_freq  DWORD',
                 i8274$INFO$6    LITERALLY   'ch_b_out_rate_port WORD,
                                              ch_b_out_rate_cmd_port WORD,
                                              ch_b_out_rate_counter  BYTE,
                                              ch_b_out_rate_freq DWORD',
                 i8274$INFO$7    LITERALLY   'ch_a_timer_type    BYTE,
                                              ch_b_timer_type    BYTE';
```

```
$subtitle('i8274$init')

/*
 *  TITLE:  i8274$init
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$init(cdata$p);
 *
 *  INTERFACE VARIABLES:
 *      cdata$p         POINTER to controller data
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      Initializes the 8274 chip.
 */

i8274$init: PROCEDURE(cdata$p) REENTRANT PUBLIC;

DECLARE
    cdata$p                                 POINTER,
    cdata   BASED   cdata$p                 TS$CDATA;

DECLARE
    i8274$info$p                            POINTER,
    i8274$info  BASED   i8274$info$p        i8274$CONTROLLER$INFO;

DECLARE
    port                                    WORD;


 /*
  *     Get the configuration info
  */

    i8274$info$p = cdata.dinfo$p;

 /*
  * Initialize driver data area (10 bytes in length)
  */

    CALL setb(0FFH, cdata.driver$cdata$p, 10);

 /*
  * Reset and Initialize the 8274.
  */

    DISABLE;
    port = i8274$info.ch_a_status_port;

    OUTPUT(port) = WR0;              /* point to WR0 */
    CALL delay(10);                  /* insure delay between outputs */
    OUTPUT(port) = CHANNEL_RESET;    /* reset channel A */
    CALL delay(10);                  /* insure delay between outputs */
    ENABLE;

    DISABLE;
    port = i8274$info.ch_b_status_port;

    OUTPUT(port) = WR0;              /* point to WR0 */
    CALL delay(10);                  /* insure delay between outputs */
    OUTPUT(port) = CHANNEL_RESET;    /* reset channel A */
    CALL delay(10);                  /* insure delay between outputs */
    ENABLE;
```

```
DISABLE;
OUTPUT(port) = WR4;             /* point to WR4 */
CALL delay(10);                 /* insure delay between outputs */
OUTPUT(port) = WR4_INIT;        /* initialize WR4 */
CALL delay(10);                 /* insure delay between outputs */
ENABLE;

DISABLE;
OUTPUT(port) = WR5;             /* point to WR5 */
CALL delay(10);                 /* insure delay between outputs */
OUTPUT(port) = WR5_TX_ENABLE;   /* initialize WR5 - Tx enabled */
CALL delay(10);                 /* insure delay between outputs */
ENABLE;

DISABLE;
OUTPUT(port) = WR3;             /* point to WR3 */
CALL delay(10);                 /* insure delay between outputs */
OUTPUT(port) = WR3_INIT;        /* initialize WR3 - Rx enabled */
CALL delay(10);                 /* insure delay between outputs */
ENABLE;

DISABLE;
OUTPUT(port) = WR1;             /* point to WR1 */
CALL delay(10);                 /* insure delay between outputs */
OUTPUT(port) = WR1_NO_INT;      /* initialize WR1 - Interrupts disabled */
CALL delay(10);                 /* insure delay between outputs */
ENABLE;

DISABLE;
port = i8274$info.ch_a_status_port;

OUTPUT(port) = WR4;             /* point to WR4 */
CALL delay(10);                 /* insure delay between OUTPUTs */
OUTPUT(port) = WR4_INIT;        /* initialize WR4 */
CALL delay(10);                 /* insure delay between OUTPUTs */
ENABLE;

DISABLE;
OUTPUT(port) = WR5;             /* point to WR5 */
CALL delay(10);                 /* insure delay between OUTPUTs */
OUTPUT(port) = WR5_TX_ENABLE;   /* initialize WR5 - Tx enabled */
CALL delay(10);                 /* insure delay between OUTPUTs */
ENABLE;

DISABLE;
OUTPUT(port) = WR3;             /* point to WR3 */
CALL delay(10);                 /* insure delay between OUTPUTs */
OUTPUT(port) = WR3_INIT;        /* initialize WR3 - Rx enabled */
CALL delay(10);                 /* insure delay between OUTPUTs */
ENABLE;

DISABLE;
OUTPUT(port) = WR1;             /* point to WR1 */
CALL delay(10);                 /* insure delay between OUTPUTs */
OUTPUT(port) = WR1_NO_INT;      /* initialize WR1 - Interrupts disabled */
CALL delay(10);                 /* insure delay between OUTPUTs */
ENABLE;

DISABLE;
port = i8274$info.ch_a_status_port;

OUTPUT(port) = WR2;             /* point to WR2 */
CALL delay(10);                 /* insure delay between OUTPUTs */
OUTPUT(port) = WR2_INIT;        /* initialize WR2 - non vectored int */
CALL delay(10);                 /* insure delay between OUTPUTs */
ENABLE;
```

```
            DISABLE;
            port = i8274$info.ch_b_status_port;

            OUTPUT(port) = WR2;             /* point to WR2 */
            CALL delay(10);                 /* insure delay between OUTPUTs */
            OUTPUT(port) = NULL_VECTOR;     /* initialize WR2 - non vectored int */
            ENABLE;

        /*
         * Set the interrrupt vector in R2B to some value, and then read it
         * back to see if the chip is really there, then set to the desired
         * value.
         */
            cdata.status = E$OK;

            OUTPUT(port) = WR2;             /* point to WR2 */
            CALL delay(10);                 /* insure delay between OUTPUTs */
            OUTPUT(port) = TEST_VECTOR;     /* interrupt vector for RR2B */

            CALL TIME(10);

            OUTPUT(port) = RR2;             /* point to RR2 */
            CALL delay(10);                 /* insure delay between OUTPUTs */

            IF (INPUT(port) AND VECTOR_MASK) <> (TEST_VECTOR AND VECTOR_MASK)
            THEN
                cdata.status = E$IO;

            CALL TIME(10);

            OUTPUT(port) = WR2;             /* point to WR2 */
            CALL delay(10);                 /* insure delay between OUTPUTs */
            OUTPUT(port) = NULL_VECTOR;     /* null interrupt vector for RR2B */

            CALL TIME(10);

            OUTPUT(port) = RR2;             /* point to RR2 */
            CALL delay(10);                 /* insure delay between OUTPUTs */

            IF (INPUT(port) AND VECTOR_MASK) <> 0
            THEN
                cdata.status = E$IO;

        END i8274$init;

        $subtitle('i8274$setup')

        /*
         *  TITLE:  i8274$setup
         *
         *  CALLING SEQUENCE:
         *      CALL i8274$setup(udata$p);
         *
         *  INTERFACE VARIABLES:
         *      udata$p     POINTER to unit data
         *
         *  CALLS:
         *      none
         *
         *  ABSTRACT:
         *      Initializes the baud rate generator to the configured
         *      rate, and sets up the 8274 for asychronous mode,
         *      divide by 16, 8 data bits, 1 stop
         *      bit; parity generation per configuration.
         */
```

```
i8274$setup: PROCEDURE(udata$p) REENTRANT PUBLIC;

DECLARE
    udata$p                                 POINTER,
    udata$p$o       STRUCTURE(
                                    offset  WORD,
                                    base    SELECTOR) AT(@udata$p),
    cdata   BASED   udata$p$o.base          TS$CDATA,
    udata   BASED   udata$p                 TS$UDATA;

DECLARE
    i8274$info$p                            POINTER,
    i8274$info  BASED   i8274$info$p        i8274$CONTROLLER$INFO;

DECLARE
    ch_p                                    POINTER,
    ch  BASED   ch_p    STRUCTURE (
                                        data_port       WORD,
                                        status_port     WORD ),
    ch_rate_p                               POINTER,
    ch_rate BASED   ch_rate_p   STRUCTURE (
                                        in_port         WORD,
                                        in_cmd_port     WORD,
                                        in_counter      BYTE,
                                        in_freq         DWORD,
                                        out_port        WORD,
                                        out_cmd_port    WORD,
                                        out_counter     BYTE,
                                        out_freq        DWORD),

    driver$data$p                           POINTER,
    driver$data BASED   driver$data$p   STRUCTURE(
                                        ch_a$in$rate    WORD,
                                        ch_a$out$rate   WORD,
                                        ch_a$parity     BYTE,
                                        ch_b$in$rate    WORD,
                                        ch_b$out$rate   WORD,
                                        ch_b$parity     BYTE);


DECLARE
    temp                                    BYTE,
    port                                    WORD,
    out_cmd                                 BYTE,
    parity$mode                             BYTE,
    timer$type                              BYTE,
    rate$count                              WORD,
    in$rate                                 WORD,
    out$rate                                WORD,
    parity                                  BYTE;


    i8274$info$p = cdata.dinfo$p;
    driver$data$p = cdata.driver$cdata$p;

    IF udata.unit$number = 0 THEN
    DO;
        ch_p = @i8274$info.ch_a_data_port;
        ch_rate_p = @i8274$info.ch_a_in_rate_port;
        timer$type = i8274$info.ch_a_timer_type;
        in$rate = driver$data.ch_a$in$rate;
        out$rate = driver$data.ch_a$out$rate;
        parity = driver$data.ch_a$parity;
    END;
    ELSE
```

```
            DO;
                ch_p = @i8274$info.ch_b_data_port;
                ch_rate_p = @i8274$info.ch_b_in_rate_port;
                timer$type = i8274$info.ch_b_timer_type;
                in$rate = driver$data.ch_b$in$rate;
                out$rate = driver$data.ch_b$out$rate;
                parity = driver$data.ch_b$parity;
            END;

            out_cmd = WR5_TX_ENABLE;

        /*
         *  Initialize the input rate generator if the baud rate has changed, or
         *  if a baud rate scan is in progress, and if it's programmable.
         */

            IF (in$rate <> udata.in$rate) AND
                (ch_rate.in_freq <> 0) AND (udata.in$rate <> HARDWARE$BAUD$SELECT)
                THEN
            DO;

                IF udata.in$rate <= AUTO$BAUD$SELECT THEN
                DO;
                    rate$count = SHR(19200, (udata.in$rate-1)*3);
                    out_cmd = WR5_TX_DISABLE;
                END;
                ELSE
                DO;
                    rate$count = udata.in$rate;
                    in$rate = udata.in$rate;
                END;

               /*
                * The initial timer value is the timer input frequency
                * divided by the configured baud rate.
                */

                temp = FALSE;
                IF (ch_rate.in_freq MOD rate$count) >= SHR(rate$count,1) THEN
                    temp = TRUE;
                rate$count = (ch_rate.in_freq / rate$count);
                IF temp THEN
                    rate$count = rate$count + 1;

                CALL set$baud$rate$count(ch_rate.in_cmd_port,
                                         ch_rate.in_port,
                                         timer$type,
                                         ch_rate.in_counter,
                                         rate$count);

            END;

        /*
         *  initialize the output baud rate generator, if there is one, and it has
         *  changed, and it's programmable.
         */

            IF (out$rate <> udata.out$rate) AND
                (ch_rate.out_freq <> 0) AND (udata.out$rate <> HARDWARE$BAUD$SELECT)
                THEN
            DO;
                IF udata.out$rate <> OUT$BAUD$SAME THEN
                DO;
                    temp = FALSE;
                    IF (ch_rate.out_freq MOD udata.out$rate) >= SHR(udata.out$rate,1)
                        THEN
                            temp = TRUE;
                    rate$count = (ch_rate.out_freq / udata.out$rate);
```

```
186   4                  IF temp THEN
187   4                      rate$count = rate$count + 1;
188   4              END;

189   3              out$rate = udata.out$rate;

                    /*
                     * The initial timer value is the timer output frequency
                     * divided by the configured baud rate.
                     */

190   3              CALL set$baud$rate$count(ch_rate.out_cmd_port,
                                              ch_rate.out_port,
                                              timer$type,
                                              ch_rate.out_counter,
                                              rate$count);

191   3          END;


             /*
              *  figure out the parity control part of the mode word.
              */

192   2          IF (udata.term$flags AND OUT$PARITY$MASK) = EVEN$OUTPUT$PARITY$MODE THEN
193   2              parity$mode = EVEN$MODE;
194   2          ELSE
                 DO;

195   3              IF (udata.term$flags AND OUT$PARITY$MASK) = ODD$OUTPUT$PARITY$MODE
                     THEN
196   3                  parity$mode = ODD$MODE;
197   3              ELSE
                         parity$mode = NO$PARITY$MODE;

198   3          END;


199   2          port = ch.status_port;

                 /*
                  *  If a new parity is specified, set up this 8274 channel accordingly.
                  */

200   2          IF parity$mode <> parity THEN
201   2          DO;
202   3              parity = parity$mode;

203   3              OUTPUT(port) = WR4;                         /* point to WR4 */
204   3              CALL delay(10);                /* insure delay between OUTPUTs */
205   3              OUTPUT(port) = WR4_INIT OR parity$mode;

206   3              CALL TIME(10);
207   3          END;

208   2          OUTPUT(port) = WR5;                         /* point to WR5 */
209   2          CALL delay(10);                /* insure delay between outputs */
210   2          OUTPUT(port) = out_cmd;

211   2          CALL TIME(10);

212   2          OUTPUT(port) = WR3;                         /* point to WR3 */
213   2          CALL delay(10);                /* insure delay between outputs */
214   2          OUTPUT(port) = WR3_INIT;
215   2          CALL delay(10);                /* insure delay between outputs */
```

```
/*
 * Throw away any chars from baud rate search.
 */
    DO WHILE (INPUT(ch.status_port) AND RX_CHAR_RDY) <> 0;
        temp = INPUT(ch.data_port);
        CALL delay(10);                 /* insure delay between outputs */
    END;

/*
 * If the 8274 is ready for output, tell the terminal support
 * to send a char.
 */

    CALL delay(10);                 /* insure delay between outputs */
    IF (INPUT(ch.status_port) AND TX_BUFFER_EMPTY) <> 0 THEN
        CALL xts$set$output$waiting(udata$p);

/*
 * Allow Tx and Rx interrupts now.
 */

    CALL delay(10);                 /* insure delay between outputs */
    OUTPUT(port) = WR1;                         /* point to WR1 */
    CALL delay(10);                 /* insure delay between outputs */
    OUTPUT(port) = WR1_INIT;


END i8274$setup;

$subtitle('i8274$check')

/*
 *  TITLE:  i8274$check
 *
 *  CALLS:
 *      none
 *
 *  INTERFACE VARIABLES:
 *      cdata$p         POINTER to controller data
 *
 *  CALLING SEQUENCE:
 *      ch = i8274$check(cdata$p);
 *
 *  ABSTRACT:
 *      Term$check procedure, connected to 8274 input interrupt.
 *      Gets input char, strips off parity if required, and sets
 *      up flags for terminal support.
 */

i8274$check:    PROCEDURE(cdata$p) BYTE REENTRANT PUBLIC;

DECLARE
    cdata$p                             POINTER,
    cdata   BASED   cdata$p             TS$CDATA;

DECLARE
    i8274$info$p                        POINTER,
    i8274$info  BASED   i8274$info$p    i8274$CONTROLLER$INFO,
    udata$p                             POINTER,
    udata$p$o       STRUCTURE (
                                offset  WORD,
                                base    SELECTOR ) AT (@udata$p),
    udata   BASED   udata$p TS$UDATA;
```

```
DECLARE
    ch_p                                    POINTER,
    ch  BASED   ch_p    STRUCTURE (
                              data_port   WORD,
                              status_port WORD ),
    ch_rate_p                               POINTER,
    ch_rate BASED   ch_rate_p  STRUCTURE (
                              in_port     WORD,
                              in_cmd_port WORD,
                              in_counter  BYTE,
                              in_freq     DWORD,
                              out_port    WORD,
                              out_cmd_port    WORD,
                              out_counter BYTE,
                              out_freq    DWORD );

DECLARE
    unit                                    BYTE,
    vector                                  BYTE,
    dummy                                   BYTE,
    found$rate                              BYTE,
    i                                       WORD,
    char                                    BYTE;

    i8274$info$p = cdata.dinfo$p;

/*
 *  find out what caused the interrupt by reading RR2B
 */
    OUTPUT(i8274$info.ch_b_status_port) = 002H;
    CALL delay(5);                  /* insure delay between outputs */
    vector = INPUT(i8274$info.ch_b_status_port);
    CALL delay(20);                 /* insure delay between outputs */

    IF ((vector AND NO_INT_VECT) = NO_INT_VECT) AND
       ((INPUT(i8274$info.ch_a_status_port) AND INT_PENDING) = 0) THEN
    DO;
        c$data.interrupt$type = NO$INTERRUPT;
        RETURN char;
    END;

    IF (vector AND 10H) = 10H THEN
    DO;
        ch_p = @i8274$info.ch_a_data_port;
        ch_rate_p = @i8274$info.ch_a_in_rate_port;
        c$data.interrupting$unit = 0;
    END;
    ELSE
    DO;
        ch_p = @i8274$info.ch_b_data_port;
        ch_rate_p = @i8274$info.ch_b_in_rate_port;
        c$data.interrupting$unit = 1;
    END;

/*
 * Set up udata$p to point to the interrupting units data.
 * ( that is, add 1024 to the pointer for each unit )
 */

    udata$p = @cdata.udata;
    udata$p$o.offset = udata$p$o.offset +
                            SHL(DOUBLE(cdata.interrupting$unit),10);

    vector = (SHR(vector,2) AND 03H);
```

```
/*
 * Modify the vector so that Special Rx Condition interrupts
 * are handled in the Rx Char. Available case.
 */

   IF vector = 3 THEN
   DO;
      vector = 2;
      OUTPUT(ch.status_port) = ERROR_RESET;
      CALL delay(2);                  /* insure delay between outputs */
   END;

   IF vector = 2 THEN
   DO;
         /* Rx Char. available */

         char = INPUT(ch.data_port);

         /*
          *  If in auto baud rate search, check character for
          *  an identifiable baud rate
          */

         IF udata.in$rate <= AUTO$BAUD$SELECT THEN
         DO;
            char = char AND 07FH;
            IF (char = 55H) THEN
               found$rate = 0;
            ELSE
            DO;
               IF char = 66H THEN
                  found$rate = 1;
               ELSE
               DO;
                  IF char = 78H THEN
                     found$rate = 2;
                  ELSE
                  DO;
                     IF char = 0 THEN
                     DO;
                     /*
                      *  Go to next baud rate range and
                      *  condition terminal support to call setup
                      *  in about 150 ms.
                      */
                        udata.in$rate = udata.in$rate + 1;
                        IF udata.in$rate > AUTO$BAUD$SELECT THEN
                           udata.in$rate = 1;
                        OUTPUT(ch.status_port) = WR1;
                        CALL delay(10); /* insure delay between outputs */
                        OUTPUT(ch.status_port) = WR1_NO_RX_INT;
                        CALL delay(10); /* insure delay between outputs */
                        cdata.interrupt$type = DELAY$INTERRUPT;
                        OUTPUT(ch.status_port) = WR3;
                        CALL delay(10); /* insure delay between outputs */
                        OUTPUT(ch.status_port) = WR3_RX_DISABLE;
                        CALL delay(10); /* insure delay between outputs */
                        /*  CALL TIME(10); */
                        OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;
                        RETURN char;
                     END;
                     ELSE
                     DO;
                        IF udata.in$rate <> 3 THEN
                        DO;
                           cdata.interrupt$type = MORE$INTERRUPT;
                           RETURN char;
                        END;
                        ELSE
                        DO;
                            udata.in$rate = 110;
                            OUTPUT(ch.status_port) = WR1;
                            CALL delay(10); /* insure delay between outputs */
                            OUTPUT(ch.status_port) = WR1_NO_RX_INT;
                            CALL delay(10); /* insure delay between outputs */
                            cdata.interrupt$type = DELAY$INTERRUPT;
                            OUTPUT(ch.status_port) = WR3;
                            CALL delay(10); /* insure delay between outputs */
                            OUTPUT(ch.status_port) = WR3_RX_DISABLE;
                            CALL delay(10); /* insure delay between outputs */
                            /*  CALL TIME(10); */
                            OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;
                            RETURN char;
                        END;
                    END;
                END;
            END;
        END;

       /*
        *   Calculate recognized baud rate
        */

        udata.in$rate = SHR(19200, (udata.in$rate-1) * 3 + found$rate);
        OUTPUT(ch.status_port) = WR1;
        CALL delay(10); /* insure delay between outputs */
        OUTPUT(ch.status_port) = WR1_NO_RX_INT;
        CALL delay(10); /* insure delay between outputs */
        cdata.interrupt$type = DELAY$INTERRUPT;
        OUTPUT(ch.status_port) = WR3;
        CALL delay(10); /* insure delay between outputs */
        OUTPUT(ch.status_port) = WR3_RX_DISABLE;
        CALL delay(10); /* insure delay between outputs */
        /* CALL TIME(10); */
        OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;
        RETURN char;
    END;

/*
 *  check input parity mode & strip parity if desired
 */

    IF (udata.term$flags AND IN$PARITY$MASK) <> PASS$INPUT$PARITY$MODE
        THEN
    DO;
        IF (udata.term$flags AND IN$PARITY$MASK) =
            STRIP$INPUT$PARITY$MODE THEN
            char = char AND 07fh;
        ELSE
        DO;
            IF (udata.term$flags AND OUT$PAR$CHECK) <> 0 THEN
            DO;
                OUTPUT(ch.status_port) = RR1; /* point to RR1 */
                CALL delay(3);  /* insure delay between outputs */
                IF (input(ch.status_port) AND i8274$INPUT$ERROR) <> 0
                    THEN
                DO;
                    char = char OR 080H;
                    OUTPUT(ch.status_port) = ERROR_RESET;
                    CALL delay(10); /* insure delay between outputs */
                    OUTPUT(ch.status_port) = WR3;
                    CALL delay(10); /* insure delay between outputs */
                    OUTPUT(ch.status_port) = WR3_INIT;
                END;
            END;
            ELSE
```

```
                    DO;
                        IF (udata.term$flags AND IN$PARITY$MASK) =
                           EVEN$INPUT$PARITY$MODE THEN
                        DO;
                            dummy = 0;
                            char = char OR dummy;
                            IF PARITY THEN
                                char = char AND 07FH;
                            ELSE
                                char = char OR 080H;
                        END;
                        ELSE
                        DO;
                            dummy = 0;
                            char = char OR dummy;
                            IF NOT PARITY THEN
                                char = char AND 07FH;
                            ELSE
                                char = char OR 080H;
                        END;
                    END;
                END;
            END;

            OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;

            cdata.interrupt$type = INPUT$INTERRUPT;

    END;
    ELSE
    DO;
        IF vector = 0 THEN
        DO; /* Tx Buffer empty */

            OUTPUT(ch.status_port) = RESET_TX_INT;
            CALL delay(5);                   /* insure delay between outputs */
            cdata.interrupt$type = OUTPUT$INTERRUPT;
            OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;

        END;
        ELSE
        DO; /* Ext/Status Change */

            OUTPUT(ch.status_port) = RESET_EXT_INT;
            cdata.interrupt$type = MORE$INTERRUPT;
            CALL delay(5);                   /* insure delay between outputs */
            OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;

        END; /* Ext/Status Change */
    END;

    RETURN char;

END i8274$check;
```

```
          $subtitle('i8274$answer')

          /*
           *  TITLE:  i8274$answer
           *
           *  CALLING SEQUENCE:
           *      CALL i8274$answer(udata$p);
           *
           *  INTERFACE VARIABLES:
           *      udata$p     POINTER to unit data
           *
           *  CALLS:
           *      none
           *
           *  ABSTRACT:
           *      Sends a mode word to the 8274 to place DTR active.
           *
           */

390   1   i8274$answer:   PROCEDURE(udata$p) REENTRANT PUBLIC;

391   2   DECLARE
              udata$p                                POINTER,
              udata$p$o   STRUCTURE(
                                      offset      WORD,
                                      base        SELECTOR) AT(@udata$p),
              cdata   BASED   udata$p$o.base   TS$CDATA,
              udata   BASED   udata$p          TS$UDATA;

392   2   DECLARE
              i8274$info$p                           POINTER,
              i8274$info  BASED   i8274$info$p    i8274$CONTROLLER$INFO;

393   2   DECLARE
              ch_p                                   POINTER,
              ch  BASED   ch_p    STRUCTURE (
                                      data_port   WORD,
                                      status_port WORD );

394   2       i8274$info$p = cdata.dinfo$p;

395   2       IF udata.unit$number = 0 THEN
396   2           ch_p = @i8274$info.ch_a_data_port;
397   2       ELSE
                  ch_p = @i8274$info.ch_b_data_port;

398   2       OUTPUT(ch.status_port) = WR5;
399   2       CALL delay(10);             /* insure delay between outputs */
400   2       OUTPUT(ch.status_port) = WR5_DTR_ON;

401   2   END i8274$answer;
```

```
$subtitle('i8274$hangup')

/*
 *  TITLE:  i8274$hangup
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$hangup(udata$p);
 *
 *  INTERFACE VARIABLES:
 *      udata$p     POINTER to unit data
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      Sends a mode word to the 8274 to place DTR inactive.
 *
 */
i8274$hangup:   PROCEDURE(udata$p) REENTRANT PUBLIC;

DECLARE
    udata$p                             POINTER,
    udata$p$o   STRUCTURE(
                            offset      WORD,
                            base        SELECTOR) AT(@udata$p),
    cdata   BASED   udata$p$o.base  TS$CDATA,
    udata   BASED   udata$p         TS$UDATA;

DECLARE
    i8274$info$p                        POINTER,
    i8274$info  BASED   i8274$info$p    i8274$CONTROLLER$INFO;

DECLARE
    ch_p                                POINTER,
    ch  BASED   ch_p    STRUCTURE (
                            data_port   WORD,
                            status_port WORD );

    i8274$info$p = cdata.dinfo$p;

    IF udata.unit$number = 0 THEN
        ch_p = @i8274$info.ch_a_data_port;

    ELSE
        ch_p = @i8274$info.ch_b_data_port;

    OUTPUT(ch.status_port) = WR5;
    CALL delay(10);             /* insure delay between outputs */
    OUTPUT(ch.status_port) = WR5_DTR_OFF;

END i8274$hangup;
```

```
$subtitle('i8274$out')

/*
 *  TITLE:  i8274$out
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$out(udata$p,char);
 *
 *  INTERFACE VARIABLES:
 *      udata$p     POINTER to unit data
 *      char        BYTE to OUTPUT
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      OUTPUTs a char to selected channel of the 8274.
 *      Marking or spacing parity is handled here if enabled,
 *      and the char is sent out.
 *
 */
i8274$out: PROCEDURE(udata$p,char) PUBLIC REENTRANT;

DECLARE
    udata$p                                 POINTER,
    udata$p$o   STRUCTURE(
                            offset      WORD,
                            base        SELECTOR) AT(@udata$p),
    cdata   BASED   udata$p$o.base  TS$CDATA,
    udata   BASED   udata$p         TS$UDATA;

DECLARE
    i8274$info$p                            POINTER,
    i8274$info  BASED   i8274$info$p    i8274$CONTROLLER$INFO;

DECLARE
    ch_p                                    POINTER,
    ch  BASED   ch_p    STRUCTURE (
                            data_port   WORD,
                            status_port WORD );

DECLARE
    char                                    BYTE,
    mode                                    WORD;

    i8274$info$p = cdata.dinfo$p;

    IF udata.unit$number = 0 THEN
        ch_p = @i8274$info.ch_a_data_port;

    ELSE
        ch_p = @i8274$info.ch_b_data_port;

    mode = udata.term$flags AND OUT$PARITY$MASK;
    IF mode <= MARK$OUTPUT$PARITY$MODE THEN
    DO;
        IF mode = MARK$OUTPUT$PARITY$MODE THEN
            char = char OR 80H;
        ELSE
            char = char AND 07FH;
    END;

    OUTPUT(ch.data_port) = char;


END i8274$out;
```

```
$subtitle('i8274$finish')

/*
 *  TITLE:  i8274$finish
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$finish(cdata$p);
 *
 *  INTERFACE VARIABLES:
 *      cdata$p - pointer to controller data.
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      Procedure disables TX, RX and interrupts.
 *
 */
i8274$finish: PROCEDURE (cdata$p) PUBLIC REENTRANT;

DECLARE
    cdata$p                                  POINTER,
    cdata   BASED   cdata$p                  TS$CDATA;

DECLARE
    i8274$info$p                             POINTER,
    i8274$info  BASED   i8274$info$p         i8274$CONTROLLER$INFO;

DECLARE
    port                                     WORD;


/*
 *      Get the configuration info
 */
   i8274$info$p = cdata.cinfo$p;

/*
 *  Disable the 8274 TX, RX, and interrupts.
 */

    port = i8274$info.ch_b_status_port;

    OUTPUT(port) = WR5;                 /* point to WR5 */
    CALL delay(10);                     /* insure delay between outputs */
    OUTPUT(port) = WR5_TX_DISABLE;      /* disable Tx
    CALL delay(10);                     /* insure delay between outputs */

    OUTPUT(port) = WR3;                 /* point to WR3 */
    CALL delay(10);                     /* insure delay between outputs */
    OUTPUT(port) = WR3_RX_DISABLE;      /* disable Rx
    CALL delay(10);                     /* insure delay between outputs */

    OUTPUT(port) = WR1;                 /* point to WR1 */
    CALL delay(10);                     /* insure delay between outputs */
    OUTPUT(port) = WR1_NO_INT;          /* disable interrupts
    CALL delay(10);                     /* insure delay between outputs */

    port = i8274$info.ch_a_status_port;

    OUTPUT(port) = WR5;                 /* point to WR5 */
    CALL delay(10);                     /* insure delay between outputs */
    OUTPUT(port) = WR5_TX_DISABLE;      /* disable Tx
    CALL delay(10);                     /* insure delay between outputs */

    OUTPUT(port) = WR3;                 /* point to WR3 */
    CALL delay(10);                     /* insure delay between outputs */
    OUTPUT(port) = WR3_RX_DISABLE;      /* disable Rx
    CALL delay(10);                     /* insure delay between outputs */

    OUTPUT(port) = WR1;                 /* point to WR1 */
    CALL delay(10);                     /* insure delay between outputs */
    OUTPUT(port) = WR1_NO_INT;          /* disable interrupts
    CALL delay(10);                     /* insure delay between outputs */

END i8274$finish;

END x8274;
```

```
   MODULE INFORMATION:

        CODE AREA SIZE     = 0943H   2371D
        CONSTANT AREA SIZE = 0000H      0D
        VARIABLE AREA SIZE = 0000H      0D
        MAXIMUM STACK SIZE = 0030H     48D
        1394 LINES READ
        0 PROGRAM WARNINGS
        0 PROGRAM ERRORS

   DICTIONARY SUMMARY:

        96KB MEMORY AVAILABLE
        22KB MEMORY USED   (22%)
        0KB DISK SPACE USED

   END OF PL/M-86 COMPILATION
/*
 *    x8255.lit
 *
 *  8255 is programmed as follows:
 *
 *    Group A:  Mode 1
 *    Group B:  Mode 0
 *
 *    Port A and Lower Port C: OUTPUT
 *    Port B and Upper Port C: INPUT
 *
 *  Port C definition (bit 0 is LSB;  bit 7 is MSB):
 *
 *     Bit  0   -   Character strobe to the printer
 *          1   -   not used
 *          2   -   not used
 *          3   -   Character acknowledge from the printer is complete
 *          4   -   Printer ready
 *          5   -   Paper out
 *          6   -   Printer interrupt enable
 *          7   -   Character acknowledge from the printer
 */
DECLARE
   MODE$WORD          LITERALLY   '0AAH',
   CHAR$ACK$COMPLETE  LITERALLY   '08H',
   PRINTER$READY      LITERALLY   '10H',
   PAPER$OUT          LITERALLY   '20H',
   CHAR$ACK           LITERALLY   '80H',
   INT$ENABLE         LITERALLY   '0DH',
   INT$DISABLE        LITERALLY   '0CH',
   STROBE$ON          LITERALLY   '01H',
   STROBE$OFF         LITERALLY   '00H';


    /*
     * xprntr.lit
     *
     * Common device driver information
     *
     * level:               Interrupt level
     * priority:            Priority of interrupt task
     * stack$size:          Stack size for interrupt task
     * data$size:           Device local data size
     * num$units:           Number of units on device
     * device$init:         Init device procedure
     * device$finish:       Finished with device procedure
     * device$start:        Start device procedure
     * device$stop:         Stop device procedure
     * device$interrupt:    Device interrupt procedure
     */

DECLARE COMMON$DEV$INFO LITERALLY
    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';
```

```
DECLARE i8255$INFO LITERALLY
    A$port              WORD,
    B$port              WORD,
    C$port              WORD,
    Control$port        WORD';



DECLARE
    PRINTER$DEVICE$INFO LITERALLY 'STRUCTURE(
        COMMON$DEV$INFO,
        i8255$INFO,
        tab$control   WORD)';


$save nolist
/*
 * x206dv.lit
 *      Defines literals for 206 driver
 *
 */

    /*
     * The iopb fields (first 9 bytes) must be first!!
     * They are used later and the other procedures
     * do not know of status or restore.
     *
     * Note that each spindle has up to 4 platters, and each 206 can support
     * up to 4 spindles.  Thus, there are 4 statuses:  one for each spindle.
     *
     * Restore is used to indicate that there is a restore in progress.
     * It is set when a restore is started after a request returns an
     * error which requires a restore to reset the drive.  A new request
     * is not started when there is a restore in progress, instead the
     * interrupt routine starts the request and resets restore when
     * the restore finishes.
     */

    DECLARE
        IO$PARM$BLOCK$206   LITERALLY 'STRUCTURE(
            inter               BYTE,
            instr               BYTE,
            r$count             BYTE,
            cyl$add             BYTE,
            rec$add             BYTE,
            buff$p              POINTER,
            status(4)           BYTE,
            restore             BYTE,
            format$table(72)    BYTE)';

    /*
     *  defines masks
     */

    DECLARE
        inter$on$mask       LITERALLY '008H',   /* bit 4 := 16-bit data */
        inter$off$mask      LITERALLY '018H',
        FORMAT$TRACK$ON     LITERALLY '040H',
        i206$TRACK$MAX      LITERALLY '800',    /* 400 tracks * 2 surfaces */
        i206$SECTOR$MAX     LITERALLY '36',
        command$busy        LITERALLY '080H';

    /*
     *  defines op-codes
     */

    DECLARE
        no$op               LITERALLY '00H',
        seek$op             LITERALLY '01H',
        format$op           LITERALLY '02H',
        restore$op          LITERALLY '03H',
        read$op             LITERALLY '04H',
        verify$op           LITERALLY '05H',
        write$op            LITERALLY '06H';
    /*
     *  defines ports
     */

    declare
        sub$system$port      LITERALLY 'base',
        result$type$port     LITERALLY 'base + 1',
        controller$stat      LITERALLY 'base + 2',
        result$byte$port     LITERALLY 'base + 3',
        inter$stat$port      LITERALLY 'base + 4',
        disk$config$port     LITERALLY 'base + 7',
        lo$seg$port          LITERALLY 'base',
        hi$seg$port          LITERALLY 'base',
        lo$off$port          LITERALLY 'base + 1',
        hi$off$port          LITERALLY 'base + 2',
        start$diagnostic     LITERALLY 'base + 5',
        reset$port           LITERALLY 'dinfo.base + 7';

$restore


$save nolist

/*
 *  x206in.lit
 *
 *  206 Driver info
 *  Adds to the device$info and unit$info structures,
 *  using common device support and random access
 *  device support.
 *
 */

/*
 *  Per device information
 */

    DECLARE
        I206$DEVICE$INFO LITERALLY 'STRUCTURE(
            RADEV$DEVICE$INFO,
            base        WORD)';

/*
 *  Per unit information
 */

    DECLARE
        I206$UNIT$INFO LITERALLY 'STRUCTURE(RAD$UNIT$INFO)';
$restore


$save nolist
    /*
     * x206dc.ext
     */

    send$206$iopb: PROCEDURE (base, iopb$p) BOOLEAN EXTERNAL;
        DECLARE
            base        WORD,
            iopb$p      POINTER;
    END send$206$iopb;

$restore
```

```
$save nolist

/*
 *  x206dp.ext
 */

    io$206: PROCEDURE (base, iors$p, duib$p, iopb$p) EXTERNAL;
        DECLARE
            base        WORD,
            iors$p      POINTER,
            duib$p      POINTER,
            iopb$p      POINTER;
    END io$206;

$restore


$save nolist

/*
 *  x206fm.ext
 */

    format$206: PROCEDURE (base, iors$p, duib$p, iopb$p) EXTERNAL;
        DECLARE
            base        WORD,
            iors$p      POINTER,
            duib$p      POINTER,
            iopb$p      POINTER;
    END format$206;

$restore


$SAVE NOLIST

/*
 *  nsleep.ext
 */

rq$sleep:  PROCEDURE( time$limit,
                      except$ptr ) EXTERNAL;

DECLARE time$limit    WORD,
        except$ptr    POINTER;

END rq$sleep;

$RESTORE


$save nolist
    /*
     * xcomon.lit
     *      Oft-used literals.
     *
     */

    DECLARE
        BOOLEAN         LITERALLY 'BYTE',
        TRUE            LITERALLY '0FFH',
        FALSE           LITERALLY '000H',
        FOREVER         LITERALLY 'WHILE TRUE',
        PTR$OVERLAY     LITERALLY 'STRUCTURE(offset WORD, base TOKEN)',
        P$OVERLAY       LITERALLY 'STRUCTURE(offset WORD, base WORD)',
        STRING          LITERALLY 'STRUCTURE(length BYTE, char(1) BYTE)',
/*
        DWORD           LITERALLY 'POINTER',
 */
        NO$TIME$LIMIT   LITERALLY '0FFFFH',
        BYTE$MAX        LITERALLY '0FFH',
        WORD$MAX        LITERALLY '0FFFFH',
        FIFO$Q          LITERALLY '000H',       /* select FIFO queueing */
        PRIO$Q          LITERALLY '001H';       /* select PRIO queueing */
$restore


$SAVE NOLIST

/*
 *  xdelay.ext
 */

/*
 * External Declaration for Delay Procedure.
 */

delay:  PROCEDURE(units) EXTERNAL;

    DECLARE
        units                           BYTE;

END delay;

$restore


$save nolist
/*
 * xdrinf.lit
 *      Driver information for common and random access devices.
 *
 */

    /*
     * Random-access driver information
     *
     * level:               Interrupt level
     * priority:            Priority of interrupt task
     * stack$size:          Stack size for interrupt task
     * data$size:           Device local data size
     * num$units:           Number of units on device
     * device$init:         Init device procedure
     * device$finish:       Finished with device procedure
     * device$start:        Start device procedure
     * device$stop:         Stop device procedure
     * device$interrupt:    Device interrupt procedure
     */

    DECLARE
        RADEV$DEVICE$INFO LITERALLY
           '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';
    /*
     * Unit info for radev
     *
     * track$size:      Size in bytes of track.  Used for calculating
     *                  track/sector.  Requests to device will not cross
     *                  track boundaries.
     * max$retry:       Number of times to retry on a soft IO error.
     */

    DECLARE
        RAD$UNIT$INFO LITERALLY
           'track$size          WORD,
            max$retry           WORD,
            cylinder$size       WORD';
$restore


$save nolist
    /*
     * xduib.lit
     *       Device-Unit Information Block definition.
     *
     */

    /*
     * name:            ASCII name of dev-unit, null padded
     * file$driver:     bit(i) ==> file-driver (i+1) is ok for this device.
     *                  See idevmg.plm
     * functs:          from EPS, bit i ==> function(i) supported by the driver.
     * flags:           For 215 only.  See EPS.
     *                  functions are F$FORMAT, F$READ, etc.
     * dev$gran:        device granularity in bytes.
     * dev$size:        size (in bytes) of device-unit
     * device:          device number/device code
     * unit:            device specific number of controller sub-unit (i.e.,
     *                  for a 204, could be 0,1 to indicate different drives)
     * dev$unit:        unique number identifying a device/unit pair for device
     *                  allocation purposes
     * init$io:         driver procedure for initializing driver
     * finish$io:       driver procedure for turning off/deallocating driver
     * queue$io:        driver procedure for queueing I/O requests
     * cancel$io:       driver procedure for cancelling I/O requests
     * device$info$p:   device specific information pointer.
     * unit$info$p:     unit specific information pointer.
     * update$timeout:  time (ticks) before update on this unit
     * num$buffers:     number of deblocking/buffering buffers for this unit
     * priority:        service task priority.
     * fixed$update:    boolean to indicate use of wall clock updates.
     * max$buffers:     maximum no. of buffers for device (used by EIOS)
     * fill:            filler byte
     */

    DECLARE
        DUIB$PART$ONE LITERALLY
           'name(DEV$NAME$LEN)  BYTE,
            file$driver     WORD,
            functs          BYTE,
            flags           BYTE,
            dev$gran        WORD,
            dev$size        DWORD,
            device          BYTE,
            unit            BYTE,
            dev$unit        WORD',
        DUIB$PART$TWO LITERALLY
           '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,
            fill            BYTE',
        DEV$UNIT$INFO$BLOCK LITERALLY 'STRUCTURE(
            DUIB$PART$ONE,
            DUIB$PART$TWO)';

    DECLARE
        VF$AUTO      LITERALLY '1',
        VF$DENSITY   LITERALLY '2',
        VF$SIDES     LITERALLY '4',
        VF$MINI      LITERALLY '8';
$restore


$save nolist
    /*
     * xexcep.lit
     *      I/O System Exception Code Mnemonics.
     */

$include(:f1:xnerro.lit)

    /*
     * IOS Synchronous Avoidable exception codes.
     */

    DECLARE
        E$NOUSER             LITERALLY '08021H',     /* Job has no Default User Object */
        E$NOPREFIX           LITERALLY '08022H';     /* Job has no Default Prefix Object */

    /*
     * IOS Asynchronous exception codes.
     */

    DECLARE
        E$FEXIST             LITERALLY '00020H',     /* File Exists */
        E$FNEXIST            LITERALLY '00021H',     /* Non-existant File */
        E$DEVFD              LITERALLY '00022H',     /* Device & File Driver Incompatable */
        E$SUPPORT            LITERALLY '00023H',     /* Un-supported Request */
        E$EMPTY$ENTRY        LITERALLY '00024H',     /* Empty Directory Entry */
        E$DIR$END            LITERALLY '00025H',     /* End of Directory */
        E$FACCESS            LITERALLY '00026H',     /* Access to File Not Granted */
        E$FTYPE              LITERALLY '00027H',     /* Bad File Type */
        E$SHARE              LITERALLY '00028H',     /* Improper File Sharing Requested */
        E$SPACE              LITERALLY '00029H',     /* No Space Left */
        E$IDDR               LITERALLY '0002AH',     /* Illegal Device Driver Request */
        E$IO                 LITERALLY '0002BH',     /* I/O Error */
        E$FLUSHING           LITERALLY '0002CH',     /* Connection is flushing requests */
        E$ILLVOL             LITERALLY '0002DH',     /* Illegal Volume */
        E$DEV$OFF$LINE       LITERALLY '0002EH',     /* Device Was Off Line */
        E$IFDR               LITERALLY '0002FH';     /* Illegal File Driver Request */

    /*
     * E$IO expanded with unitstatus codes
     */
    DECLARE
        E$IO$UNCLASS        LITERALLY '00050H',     /* Unclassified */
        E$IO$SOFT           LITERALLY '00051H',     /* Soft error */
        E$IO$HARD           LITERALLY '00052H',     /* Hard error */
        E$IO$OPRINT         LITERALLY '00053H',     /* Operator intervention required */
        E$IO$WRPROT         LITERALLY '00054H',     /* Write protected */
        E$IO$NO$DATA        LITERALLY '00055H',     /* No further data */
        E$IO$MODE           LITERALLY '00056H';     /* Mode violation */

$restore


$save nolist

/*
 *  xioexc.lit
 */

/*
 *  IO exception codes
 */

    DECLARE
        IO$UNCLASS      LITERALLY '0',
        IO$SOFT         LITERALLY '1',
        IO$HARD         LITERALLY '2',
        IO$OPRINT       LITERALLY '3',
        IO$WRPROT       LITERALLY '4',
        IO$NO$DATA      LITERALLY '5',
        IO$MODE         LITERALLY '6';
$restore


$save nolist
    /*
     * xiofct.lit
     *
     *  IO function codes
     *
     */

    DECLARE
        F$READ          LITERALLY '0',
        F$WRITE         LITERALLY '1',
        F$SEEK          LITERALLY '2',
        F$SPECIAL       LITERALLY '3',
        F$ATTACH$DEV    LITERALLY '4',
        F$DETACH$DEV    LITERALLY '5',
        F$OPEN          LITERALLY '6',
        F$CLOSE         LITERALLY '7',
        F$GETCS         LITERALLY '8',
        F$GETFS         LITERALLY '9',
        F$GETEXT        LITERALLY '10',
        F$SETEXT        LITERALLY '11',
        F$NULL$CH$ACCESS LITERALLY '12',
        F$NULL$DELETE   LITERALLY '13',
        F$RENAME        LITERALLY '14',
        F$GET$PATH$COMP LITERALLY '15',
        F$GET$DIR$ENTRY LITERALLY '16',
        F$TRUNC         LITERALLY '17',
        F$DETACH        LITERALLY '18',
        F$NUM$FUNCT     LITERALLY '19';
```

```
/*
 * Function codes for internal use only.
 * The rq$common$attach and common$io$task use F$ATTACH$THRU.
 * The req$update and common$io$task use F$UPDATE.
 */

    DECLARE
        F$ATTACH$THRU   LITERALLY '19',
        F$UPDATE        LITERALLY '20';

$restore


$save nolist
    /*
     * xiotyp.lit
     *     RMX/86 I/O System "type" literals.
     *
     */

    DECLARE
        CONNECTION  LITERALLY 'TOKEN',
        USER        LITERALLY 'TOKEN',
        BLOCK$NUM   LITERALLY '(3) BYTE';
$restore


$save nolist
    /*
     * xiors.lit
     *     I/O Request/Result Segment
     *
     */

    DECLARE
        IORS$PART$ONE LITERALLY
           'status      WORD,
            unit$status WORD,
            actual      WORD,
            actual$fill WORD,
            device      WORD,
            unit        BYTE,
            funct       BYTE,
            subfunct    WORD,
            dev$loc     DWORD,
            buff$p      POINTER',

        IORS$PART$TWO LITERALLY
           'count       WORD,
            count$fill  WORD,
            aux$p       POINTER,
            link$for    POINTER,
            link$back   POINTER,
            resp$mbox   MAILBOX,
            done        BOOLEAN,
            iors$fill   BYTE,
            cancel$id   TOKEN,
            conn$t      TOKEN',

        IO$REQ$RES$SEG  LITERALLY 'STRUCTURE(
            IORS$PART$ONE,
            IORS$PART$TWO)';

    /*
     * Define number of actual bytes of data (i.e., before links)
     */

    DECLARE
        IORS$DATA$SIZE  LITERALLY '30';
```

```
$include(:f1:xiofct.lit)
$restore


$save nolist

/*
 *  xnotif.ext
 *
 *  External for notify support procedure
 *  Called by random access supported drivers
 */

notify: PROCEDURE(unit, ddata$p) EXTERNAL;
    DECLARE
        unit        BYTE,
        ddata$p     POINTER;
END notify;

$restore


$save nolist

/*
 *  xnerro.lit
 */

    DECLARE
        E$OK                LITERALLY '00000H',
        E$TIME              LITERALLY '00001H',
        E$MEM               LITERALLY '00002H',
        E$BUSY              LITERALLY '00003H',
        E$LIMIT             LITERALLY '00004H',
        E$CONTEXT           LITERALLY '00005H',
        E$EXIST             LITERALLY '00006H',
        E$STATE             LITERALLY '00007H',
        E$NOT$CONFIGURED    LITERALLY '00008H';

    DECLARE
        E$ZERO$DIVIDE       LITERALLY '08000H',
        E$OVERFLOW          LITERALLY '08001H',
        E$TYPE              LITERALLY '08002H',
        E$BOUNDS            LITERALLY '08003H',
        E$PARAM             LITERALLY '08004H',
        E$BAD$CALL          LITERALLY '08005H';

$restore

$save nolist
    /*
     * xnutyp.lit
     *      RMX/86 Nucleus "type" literals.
     *
     */

    DECLARE
        TOKEN       LITERALLY 'SELECTOR',
        SEGMENT     LITERALLY 'TOKEN',
        TASK        LITERALLY 'TOKEN',
        REGION      LITERALLY 'TOKEN',
        SEMAPHORE   LITERALLY 'TOKEN',
        MAILBOX     LITERALLY 'TOKEN',
        JOB         LITERALLY 'TOKEN',
        EXTENSION   LITERALLY 'TOKEN';

    DECLARE
        T$MAILBOX   LITERALLY '03H',
        T$SEGMENT   LITERALLY '06H';
$restore
```

```
$save nolist
    /*
     * xparam.lit
     *      I/O System parameter literals.
     *
     */

    DECLARE
        DEV$NAME$LEN    LITERALLY '14',     /* device name is 14 bytes */
        PATH$COMP$LEN   LITERALLY '14',     /* path component size */
        UP$COMP         LITERALLY '''^''',  /* "up" component character */
        PATH$SEP        LITERALLY '''/''',  /* path component seperator character */
        DEF$PREFIX$CHAR LITERALLY '''$''';  /* default-prefix character */

    DECLARE
        ATT$DEV$TASK$STACK$SIZE         LITERALLY '512',
        CONN$JOB$DELETE$TASK$STACK$SIZE LITERALLY '512',
        TIMER$TASK$STACK$SIZE           LITERALLY '512',
        COMMON$DRIVER$STACK$SIZE        LITERALLY '512';

    DECLARE
        IOS$OS$EXTENSION    LITERALLY '192';        /* OS extension vector */

    DECLARE
        XFACE$Q$LEN         LITERALLY '(5*2)',      /* xface mbox queue length = 5*4 */
        CONN$DEL$Q$LEN      LITERALLY '(5*2)';      /* conn job-del mbox queue length = 5*4 */
$restore


$save nolist

/*
 *  xprerr.lit
 */

/*
 *  error codes
 */
DECLARE
   E$OK         LITERALLY '0000H',
   E$IDDR       LITERALLY '002AH';
$restore


$save nolist

/*
 *  xtrsec.lit
 */

DECLARE TRACK$SECTOR$STRUCT LITERALLY 'STRUCTURE(
    sector  WORD,
    track   WORD)';
$restore


$save nolist

/*
 *  xtssow.ext
 */

xts$set$output$waiting: PROCEDURE(udata$p) EXTERNAL;
DECLARE
    udata$p             POINTER;
END xts$set$output$waiting;

$restore
```

```
$SAVE NOLIST

/*
 *  xtstim.ext
 */

/*
 * External Declaration
 * for timer support procedure.
 */

set$baud$rate$count: PROCEDURE(command_port, counter_port, timer_type,
                        counter_number, rate_count) EXTERNAL;

DECLARE
    (command_port, counter_port, rate_count)    WORD,
    (timer_type, counter_number)                BYTE;

END set$baud$rate$count;

$RESTORE


$save nolist
    /*
     * xradsf.lit
     *      Random-Access driver Special-Function Mnemonics.
     *
     */

    DECLARE
        FS$FORMAT$TRACK LITERALLY '0';      /* format a track */

    /*
     *  Format info structure to format one track on
     *  a disk(hard or floppy)
     *  used by 204 & 206 drivers
     *
     */

    DECLARE
        FORMAT$INFO$STRUCT  LITERALLY 'STRUCTURE(
            track$num           WORD,
            track$interleave    WORD,
            track$skew          WORD,
            fill$char           BYTE)';

    /*
     *  Device label special function.  Asks driver to supply
     *  device information for named file label.
     *
     */

    DECLARE
        FS$DEVICE$LABEL     LITERALLY '3';

    /*
     *  Special tape functions.
     *
     */

    DECLARE
        FS$REWIND           LITERALLY '7',
        FS$READ$FILE$MARK   LITERALLY '8',
        FS$WRITE$FILE$MARK  LITERALLY '9',
        FS$RETENSION        LITERALLY '10';

$restore
```

***

INDEX

Primary references are underscored.

***

iRMX™ 86 PROGRAMMING TECHNIQUES

CONTENTS

PAGE

CONTENTS (continued)

***

CHAPTER 1 SELECTING A PL/M-86 SIZE CONTROL

This chapter applies to you only if you have decided to program your iRMX 86 tasks using PL/M-86. In order to understand the following explanation, you should be familiar with

- The PL/M-86 programming language
- PL/M-86 models of segmentation
- iRMX 86 jobs, tasks, and segments

PURPOSE OF THIS CHAPTER

Whenever you invoke the PL/M-86 Compiler, you must specify (either explicitly or by default) a program size control (SMALL, COMPACT, MEDIUM, or LARGE). This size control determines which model of segmentation the compiler uses and, consequently, greatly affects the amount of memory required to store your application's object code.

The following section explains which size control to use in order to produce the smallest object program while still satisfying the requirements of your system.

MAKING THE SELECTION

When you compile your programs using the PL/M-86 SMALL control, all POINTER values are 16 bits long. This leads to a number of restrictions, including the inability to address the contents of an iRMX 86 segment that has been received from another job. Because of these restrictions, the iRMX 86 Operating System is currently not compatible with PL/M-86 procedures compiled using the SMALL size control.

Since you cannot use the SMALL size control, you must choose between COMPACT, MEDIUM and LARGE. The algorithm for selecting a size control is presented later in this chapter. However, before you examine the algorithm, you should be aware that your choice can place restrictions on your system.

RAMIFICATIONS OF YOUR SELECTION

If you decide to use the COMPACT or MEDIUM size controls, the capabilities of your system will be slightly restricted. Only the LARGE size control preserves all of the features of the system.

Restrictions Associated With Compact

If you decide to use PL/M-86 COMPACT, you will not be able to use exception handlers. However, you can still process exceptional conditions by dealing with them in your task's code.

Restrictions Associated With Medium

If you decide to use PL/M-86 MEDIUM, you lose the option of having the iRMX 86 Operating System dynamically allocate stacks for tasks that are created dynamically. This means that you must anticipate the stack requirements of each such task, and you must explicitly reserve memory for each stack during the process of configuring the system.

DECISION ALGORITHM

Before you attempt to use the flowchart (Figure 1-1) to make your decision, note that three of the boxes are numbered. Each of these three boxes asks you to derive a quantity that represents a memory requirement of your iRMX 86 job. In order to derive the quantity requested in each of the boxes, follow the directions provided below in the section having the same number as the box.

1. COMPUTE MEMORY REQUIREMENTS FOR STATIC DATA

   Box 1 asks for an estimate of the amount of memory required to store the static data for all the tasks of your iRMX 86 job. Static data consists of all variables other than:

   - parameters in a procedure call

   - variables local to a reentrant PL/M-86 procedure

   - PL/M-86 structures that are declared to be BASED

   To obtain an accurate estimate of this quantity, use the COMPACT size control to compile the code for each task in your job. For each compilation, find the MODULE INFORMATION area at the end of the listing. Within this area is a quantity labeled VARIABLE AREA SIZE and another labeled CONSTANT AREA SIZE.

   Now you must compute the static data size for each individual compilation by adding the VARIABLE AREA SIZE to the CONSTANT AREA SIZE.

   Once you have computed the static data size for each compilation in the job, add them to obtain the static data size for the entire job.

2. COMPUTE MEMORY REQUIREMENTS FOR CODE

Box 2 asks for an estimate of the amount of memory required to store the code for all the tasks of your iRMX 86 job. To obtain this estimate, perform the following steps:

- Using the COMPACT size control, compile the code for each task in your job.
- For each compilation, find the MODULE INFORMATION area at the end of the listing. In this area is a value labeled CODE AREA SIZE. This value is the amount of memory required to store the code generated by this individual compilation.
- Sum the code requirements for all the compilations in the job. The result is the code requirement for the entire job.

3. COMPUTE MEMORY REQUIREMENTS FOR STACK

Box 3 asks for an estimate of the amount of memory required to store the stacks of all the tasks in your iRMX 86 job. If you plan to have the iRMX 86 Operating System create your stacks dynamically, your stack requirement (for the purpose of the flowchart) is zero.

If, on the other hand, you plan to create the stacks yourself, you can estimate the memory requirements by performing the following steps. Refer to the MODULE INFORMATION AREA of the compilation listings that you obtained while working with Box 2. Within this area is a value labeled MAXIMUM STACK SIZE. To this number, add the system stack requirement that you can determine by following the procedure in Chapter 8 of this manual. The result is an estimate of the stack requirement for one compilation. To compute the requirements for the entire job, just sum the requirements for all the compilations in the job.

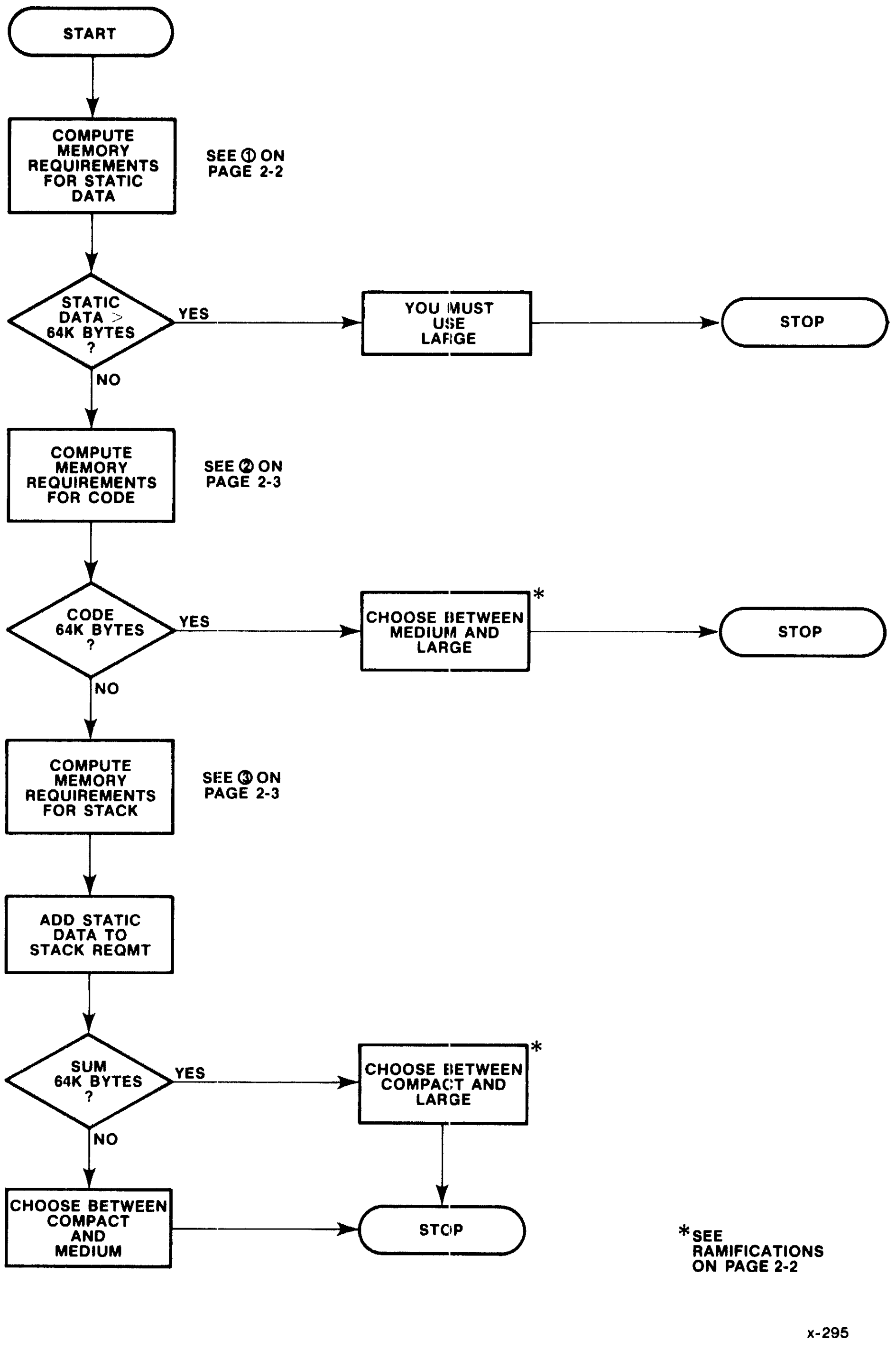

START
COMPUTE MEMORY REQUIREMENTS FOR STATIC DATA
SEE ① ON PAGE 2-2
STATIC DATA > 64K BYTES ?
YES
NO
YOU MUST USE LARGE
STOP
COMPUTE MEMORY REQUIREMENTS FOR CODE
SEE ② ON PAGE 2-3
CODE 64K BYTES ?
YES
NO
CHOOSE BETWEEN MEDIUM AND LARGE *
STOP
COMPUTE MEMORY REQUIREMENTS FOR STACK
SEE ③ ON PAGE 2-3
ADD STATIC DATA TO STACK REQMT
SUM 64K BYTES ?
YES
NO
CHOOSE BETWEEN COMPACT AND LARGE *
CHOOSE BETWEEN COMPACT AND MEDIUM
STOP
*SEE RAMIFICATIONS ON PAGE 2-2
x-295

Figure 1-1. Decision Algorithm For Size Control

***

CHAPTER 2
INTERFACE PROCEDURES AND LIBRARIES

This chapter is for anyone who writes programs that use iRMX 86 system calls. In order to understand this chapter, you should be familiar with the following concepts:

- the notion of system call
- the process of linking object modules
- the notion of an object library
- PL/M-86 size control

PURPOSE OF THIS CHAPTER

Familiarity with interface procedures is a prerequisite to understanding several of the programming techniques discussed later in this manual. The primary purpose of this chapter is to define the concept of an interface procedure and explain how it is used in the iRMX 86 Operating System.

DEFINITION OF INTERFACE PROCEDURE

The iRMX 86 Operating System uses interface procedures to simplify the process of calling one software module from another. In order to illustrate the usefulness of interface procedures, let's examine what happens without them.

Suppose you are writing an application task that will run in some hypothetical operating system. Figure 2-1 shows your application task calling two system procedures. If the system calls are direct (without an interface procedure serving as an intermediary), the application task must be bound to the system procedures either during compilation or during linking. Such binding causes your application task to be dependent upon the memory location of the system procedures.

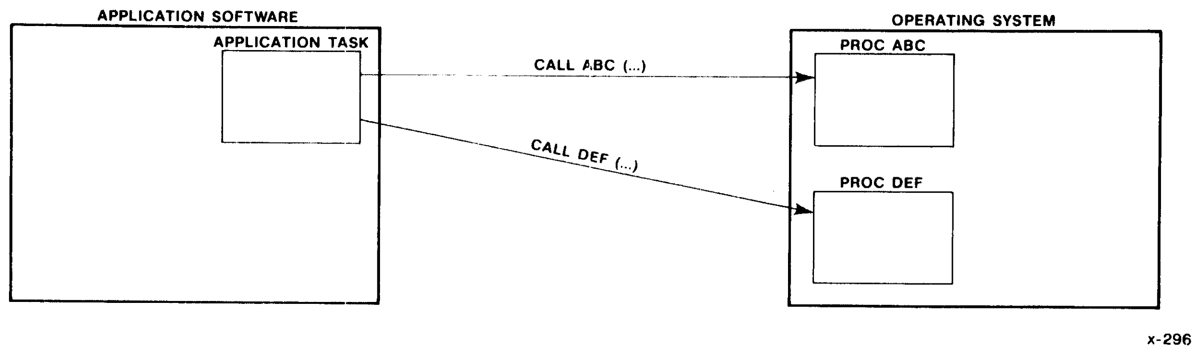


Figure 2-1. Direct Location-Dependent Invocation

Now suppose that someone updates your operating system. If, during the process of updating the system, some of the system procedures are moved to different memory locations, then your application software must be relinked to the new operating system.

There are techniques for calling system procedures that do not assume unchanging memory locations. However, most of these techniques are complex (Figure 2-2) and assume that the application programmer is intimately familiar with the interrupt architecture of the processor.

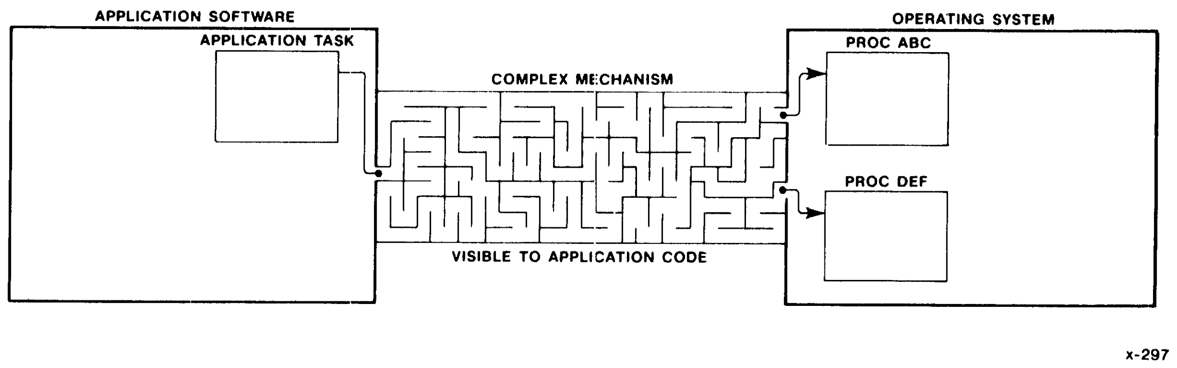


Figure 2-2. Complex Location-Independent Invocation

The iRMX 86 Operating System uses interface procedures to mask the details of location-independent invocation from the application software (Figure 2-3). Whenever application programmers need to call a system procedure from application code, they use a simple procedure call (known as a system call). This system call invokes an interface procedure which, in turn, invokes the actual system procedure.

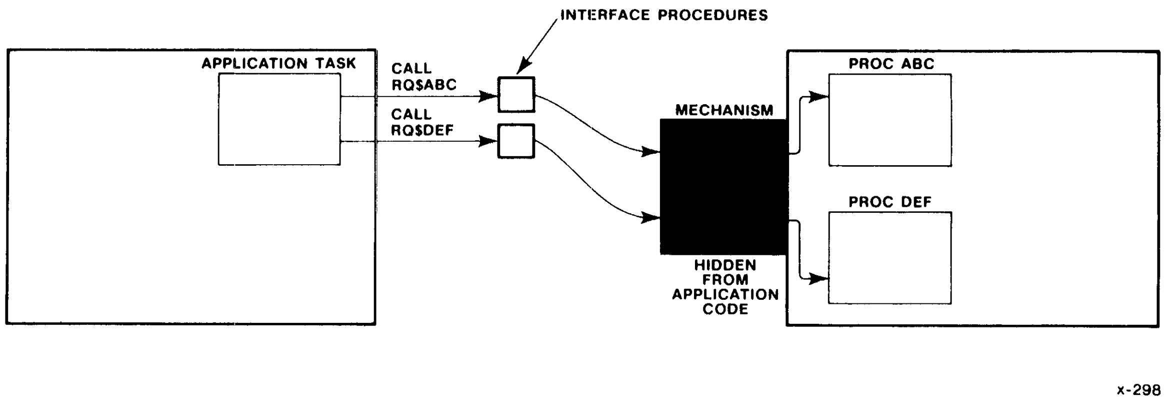


Figure 2-3. Simple Invocation Using An Interface Procedure

INTERFACE LIBRARIES

The iRMX 86 Operating System provides you with a set of object code libraries containing PL/M-86 interface procedures. These procedures preserve address independence while allowing you to invoke system calls as simple PL/M-86 procedures.

During the process of configuring an application system you must link your application software to the proper object libraries. Table 2-1 shows the correlation between subsystems of the iRMX 86 Operating System, the PL/M-86 size control, and the interface libraries. To find out which libraries you must link to, find the column that specifies the PL/M-86 size control that you are using, and the rows that specify the subsystems of the iRMX 86 Operating System that you are using. You must link to the libraries that are named at the intersections of the column and the rows.

Table 2-1. Interface Libraries and iRMX™ 86 Subsystems

	COMPACT	LARGE OR MEDIUM
NUCLEUS	RPIFC.LIB	RPIFL.LIB
BASIC I/O SYSTEM	IPIFC.LIB	IPIFL.LIB
EXTENDED I/O SYSTEM	EPIFC.LIB	EPIFL.LIB
APPLICATION LOADER	LPIFC.LIB	LPIFL.LIB
HUMAN INTERFACE	HPIFC.LIB	HPIFL.LIB
THE UNIVERSAL DEVELOPMENT INTERFACE	COMPAC.LIB	LARGE.LIB

***

CHAPTER 3
TIMER ROUTINES

This chapter is for anyone who writes programs that must determine approximate elapsed time. In order to make use of this chapter, you should be familiar with the following concepts:

- INCLUDE files
- iRMX 86 interface procedures
- iRMX 86 tasks
- initialization tasks
- using the LINK86 utility

Furthermore, if you want to understand how the timer routines work, you must be fluent in PL/M-86 and know how to use iRMX 86 regions.

PURPOSE OF THIS CHAPTER

The iRMX 86 Basic I/O System provides GET$TIME and SET$TIME system calls. These two calls supply your application with a timer having units of one second. However, if your application requires no features of the Basic I/O System other than the timer, you can reduce your memory requirements by dropping the Basic I/O System altogether and implementing the timer in your application.

This chapter provides the source code needed to build a timer into your application.

PROCEDURES IMPLEMENTING THE TIMER

Four PL/M-86 procedures are used to implement the timer. In brief, the procedures are:

- get_time

  This procedure requires no input parameter and returns a double word (POINTER) value equal to the current contents of the timer in seconds. This procedure can be called any number of times.

- set_time

  This procedure requires a double word (POINTER) input parameter that specifies the value (in seconds) to which you want the timer set. This procedure can be called any number of times.

- init_time

  This procedure creates the timer, initializes it to zero seconds, and starts it running. This procedure requires as input a POINTER to the WORD which is to receive the status of the initialization. This status will be zero if the timer is successfully created and nonzero otherwise. This procedure should be called only once.

- maintain_time

  This procedure is not called directly by your application. Rather, it runs as an iRMX 86 task that is created when your application calls init_time. The purpose of this task is to increment the contents of the timer once every second.

RESTRICTIONS

There are two important restrictions that you should keep in mind when using the timer routines:

CALL init_time FIRST

Before calling set_time or get_time, your application must call init time. You can accomplish this by calling the init_time procedure from your job's initialization task.

ONLY ONE TIMER

These procedures implement only one timer. They do not allow you to maintain a different timer for each of several purposes. For example, if one job changes the contents of the timer (by using the set_time procedure), all jobs accessing the timer will be affected.

SOURCE CODE

You can compile the following PL/M-86 source code as a single module. This will yield an object module that you can link to your application code. However, before compiling these procedures, you must create files containing the external procedure declarations for the iRMX 86 interface procedures. The names of these files are specified in the $INCLUDE statements below.

```
$title('INDEPENDENT TIMER PROCEDURES')
/*************************************************************************
*                                                                        *
*   This module consists of four procedures which implement a timer      *
*   having one-second granularity.  The outside world has access to only *
*   three of these procedures-                                           *
*                                                                        *
*       init_time                                                        *
*       set_time                                                         *
*       get_time                                                         *
*                                                                        *
*   The fourth procedure, maintain_time, is invoked by init_time and     *
*   is run as an iRMX 86 task to measure time and increment the time     *
*   counter.                                                             *
*************************************************************************/

timer:  DO;

/*************************************************************************
*   The following LITERALLY statements are used to improve the           *
*   readability of the code.                                             *
*************************************************************************/

        DECLARE
            FOREVER                 LITERALLY 'WHILE OFFH',
            DWORD                   LITERALLY 'POINTER',
            TOKEN                   LITERALLY 'WORD',
            REGION                  LITERALLY 'TOKEN',
            E$OK                    LITERALLY '00000H',
            PRIORITY_QUEUE          LITERALLY '1',
            TASK                    LITERALLY 'TOKEN';
```

```
/*************************************************************************
*                                                                        *
*   The following INCLUDE statements cause the external procedure       *
*   declarations for some of the iRMX 86 system calls to be included     *
*   in the source code.                                                  *
*                                                                        *
*************************************************************************/

$INCLUDE(:fl:icrtas.ext)  /* rq$create$task interface proc.*/
$INCLUDE(:fl:icrreg.ext)  /* rq$create$region   "       "  */
$INCLUDE(:fl:isleep.ext)  /* rq$sleep           "       "  */
$INCLUDE(:fl:idereg.ext)  /* rq$delete$region   "       "  */
$INCLUDE(:fl:iregio.ext)  /* rq$send$control    "       "  */
                     /* and  rq$receive$control "       "  */



$subtitle('Local Data')
/*************************************************************************
*   The following variables can be accessed by all of the procedures    *
*   in this module.                                         *
*************************************************************************/

        DECLARE
            time_region        REGION,       /* Guards access to time_in
                                                sec.*/

            time_in_sec        DWORD,        /* Contains time in seconds.*/

                                             /* Overlay         */
            time-in_sec_o      STRUCTURE(    /* used to obtain */
                               low   WORD,   /* high and low    */
                               high  WORD)   /* order words.    */
                               AT (@time_in_sec),

            data_seg_p         POINTER,      /* Used to obtain loc of data
                                                seg.*/

            data_seg_p_o       STRUCTURE(    /* Overlay used to */
                                offset WORD,/* obtain loc of   */
                                base   WORD)/* data segment.   */
                                AT (@data_seg_p);
```

```
$subtitle('Time maintenance task')
/*************************************************************************
*   maintain_time                                                        *
*                                                                        *
*     This procedure is run as an iRMX 86 task.  It repeatedly           *
*     performs the following algorithm-                                  *
*                                                                        *
*           Sleep 1 second.                                              *
*           Gain exclusive access to time_in_sec.                        *
*           Add 1 to time_in_sec.                                        *
*           Surrender exclusive access to time_in_sec.                   *
*                                                                        *
*     If the last three steps in the preceding algorithm require         *
*     more than one nucleus time unit, the time_in_sec counter           *
*     will run slow.                                                     *
*                                                                        *
*     This procedure must not be called by any procedure other than      *
*     init_time.                                                         *
*************************************************************************/

maintain_time: PROCEDURE REENTRANT;
    DECLARE    status  WORD;

timer_loop:
     DO FOREVER;

        CALL rq$sleep( 100, @status );  /* Sleep for one
                                           second. */

        CALL rq$receive$control         /* Gain exclusive */
                (time_region, @status); /* access.        */

        time_in_sec_o.low =             /* Add 1 second */
                time_in_sec_o.low +1;   /* to low order */
                                        /* half of timer.*/

        IF (time_in_sec_o.low = 0)      /* Handle overflow.*/
            THEN time_in_sec_o.high =
                 time_in_sec_o.high + 1;

        CALL rq$send$control(@status);  /* Surrender access*/

     END timer_loop;
END maintain_time;
```

```
$subtitle('Get Time')
/***************************************************************************
*   get time                                                               *
*                                                                          *
*     This procedure is called by the application code in order to       *
*     obtain the contents of time_in_sec.  This procedure can be          *
*     called any number of times.                                         *
***************************************************************************/

get_time: PROCEDURE DWORD REENTRANT PUBLIC;

    DECLARE   time   DWORD,
              status WORD;

    CALL rq$receive$control              /* Gain exclusive */
             ( time_region, @status);    /* access.        */

    time = time_in_sec;

    CALL rq$send$control(@status);       /* Surrender access.*/

    RETURN( time );

END get_time;



$subtitle('Set Time')
/***************************************************************************
*   set_time                                                               *
*                                                                          *
*     Application code can use this procedure to place a specific         *
*     double word value into time_in_sec.  This procedure can be          *
*     called any number of times.                                         *
***************************************************************************/

set_time: PROCEDURE( time ) REENTRANT PUBLIC;

    DECLARE time    DWORD,
            status  WORD;

    CALL rq$receive$control           /* Gain exclusive access.*/
            (time_region, @status);

    time_in_sec = time;               /* Set new time. */

    CALL rq$send$control(@status);    /* Surrender access. */

END set_time;
```

```
$subtitle('Initialize Time')
/************************************************************************
*   init_time                                                           *
*                                                                       *
*     This procedure zeros the timer, creates a task to                 *
*     maintain the timer, and a region to ensure exclusive              *
*     access to the timer.  This procedure must be called               *
*     before the first time that get_time or set_time is                *
*     called.  Also, this procedure should be called only               *
*     once.  The easiest way to make sure this happens is to            *
*     call init_time from your initialization task.                     *
*                                                                       *
*     The timer task will run in the job from which this                *
*     procedure is called.                                              *
*                                                                       *
*     If your application experiences a lot of interrupts,              *
*     the timer may run slow.  You can rectify this                     *
*     problem by raising the priority of the timer                      *
*     task.  To do this, change the 128 in the                          *
*     rq$create$task system call to a smaller number.                   *
*     This change may slow the processing of your                       *
*     interrupts.                                                       *
************************************************************************/

init_time: PROCEDURE(ret_status_p) REENTRANT PUBLIC;

    DECLARE ret_status_p       POINTER,
            ret_status         BASED ret_status_p WORD,
            timer_task_t       TASK,
            local_status       WORD;

    time_in_sec = 0;

    time_region = rq$create$region      /* Create a region. */
                     (PRIORITY_QUEUE, ret_status_p);

    IF (ret_status    E$OK) THEN
        RETURN;                         /* Return w/ error. */

    data_seg_p = @data_seg_p;           /* Get contents of
                                           DS register. */

    timer_task_t = rq$create$task       /* Create timer task. */
                    (128,               /*  priority          */
                     @maintain_time,    /*  start addr        */
                     data_seg_p_o.base, /*  data seg base     */
                     0,                 /*  stack ptr         */
                     512,               /*  stack size        */
                     0,                 /*  task flags        */
                     ret_status_p);
    IF (ret_status    E$OK) THEN
      CALL rq$delete$region                    /* Since could not */
           (time_region, @local_status);  /* create task,    */
                                               /* must delete     */
                                               /* region.         */

END init_time;

END timer;
```

***

CHAPTER 4
ASSEMBLY LANGUAGE SYSTEM CALLS

This chapter is for anyone who wants to use iRMX 86 system calls from programs written in ASM86 assembly language. In order to be able to use system calls from assembly language, you should be familiar with the following concepts:

- iRMX 86 system calls
- iRMX 86 interface procedures
- PL/M-86 size controls

You should also be familiar with PL/M-86 and fluent in ASM86 assembly language.

PURPOSE OF THIS CHAPTER

The purpose of this chapter is twofold. First, it briefly outlines the process involved in using an iRMX 86 system call from an assembly language program. Second, it directs you to other Intel manuals that provide either background information or details concerning interlanguage procedure calls.

CALLING THE SYSTEM

If you read Chapter 2 of this manual, you found that your programs communicate with the iRMX 86 System by calling interface procedures that are designed for use with programs written in PL/M-86. So the problem of using system calls from assembly language programs becomes the problem of making your assembly language programs obey the procedure-calling protocol used by PL/M-86. For example, if your ASM86 program uses the SEND$MESSAGE system call, then you must call rq$send$message interface procedure from your assembly language code.

NOTE

The techniques for calling PL/M-86 procedures from assembly language are completely described in the manual ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS for 8086-BASED DEVELOPMENT SYSTEMS.

SELECTING A SIZE CONTROL

Before writing assembly language routines that call PL/M-86 interface procedures, you must select a size control (COMPACT, MEDIUM, or LARGE) because conventions for making calls depend upon the model of segmentation.

If all of your application is written in assembly language, you can arbitrarily select a size control and use the libraries for the selected control. However, you can obtain a size and performance advantage by using the COMPACT interface procedures, since their procedure calls are all NEAR. The LARGE interface, which has procedures that require FAR procedure calls, is only advantageous if your application code is larger than 64K bytes.

On the other hand, if some of your application code is written in PL/M-86, your assembly language code should use the same interface procedures as are used by your PL/M-86 code.

***

CHAPTER 5 COMMUNICATION BETWEEN iRMX™ 86 JOBS

This chapter applies to anyone who wants to pass information from one iRMX 86 job to another. In order to understand this chapter, you must be familiar with the following concepts:

- iRMX 86 jobs, including object directories
- iRMX 86 tasks
- iRMX 86 segments
- the root job of an iRMX 86-based system
- iRMX 86 mailboxes
- iRMX 86 physical files or named files
- iRMX 86 stream files
- iRMX 86 type managers and composite objects

PURPOSE OF THIS CHAPTER

In multiprogramming systems, where each of several applications is implemented as a distinct iRMX 86 job, there is an occasional need to pass information from one job to another. This chapter describes several techniques that you can use to accomplish this.

The techniques are divided into two collections. The first collection deals with passing large amounts of information from one job to another, while the second collection deals with passing iRMX 86 objects.

PASSING LARGE AMOUNTS OF INFORMATION BETWEEN JOBS

There are three methods for sending large amounts of information from one job to another:

1) You can create an iRMX 86 segment and place the information in the segment. Then, using one of the techniques discussed below for passing objects between jobs, you can deliver the segment.

The advantages of this technique are:

- Since this technique requires only the Nucleus, you can use it in systems that do not use other iRMX 86 subsystems.
- The iRMX 86 Operating System does not copy the information from one place to another.

The disadvantages of this technique are:

- The segment will occupy memory until it is deleted, either explicitly (by means of the DELETE$SEGMENT system call), or implicitly (when the job that created the segment is deleted). Until the segment is deleted, a substantial amount of memory is unavailable for use elsewhere in the system.
- The application code may have to copy the information into the segment.

2) You can use an iRMX 86 stream file.

The advantages of this technique are:

- The data need not be broken into records.
- This technique can easily be changed to Technique 3.

The disadvantage of this technique is that you must configure one or both I/O systems into your application system.

3) You can use either the Extended or the Basic I/O System to write the information onto a mass storage device, from which the job needing the information can read it.

The advantages of this technique are:

- Many jobs can read the information.
- This technique can easily be changed to Technique 2.
- The information need not be divided into records.

The disadvantages of this technique are:

- You must incorporate one or both I/O systems into your application system.
- Device I/O is slower than reading and writing to a stream file.

PASSING OBJECTS BETWEEN JOBS

Jobs can also communicate with each other by sending objects across job boundaries. You can use any of several techniques to accomplish this, and you should avoid using one seemingly straightforward technique. In the following discussions you will see how to pass objects by using object directories, mailboxes, and parameter objects. You will also see why you should not pass object tokens by embedding them in an iRMX 86 segment or in a fixed memory location.

Although you can pass any object from one job to another, there is a restriction pertaining to connection objects. When a file connection created in one job (Job A) is passed to a second job (Job B) the second job (Job B) cannot successfully use the object to perform I/O. Instead, the second job (Job B) must create another connection to the same file. This restriction is discussed in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL and in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

PASSING OBJECTS THROUGH OBJECT DIRECTORIES

For the purpose of this discussion, consider a hypothetical system in which tasks in separate jobs must communicate with each other. Specifically, suppose that Task B in Job B must not begin or resume running until Task A in Job A grants permission.

One way to perform this synchronization is to use a semaphore. Task B can repeatedly wait at the semaphore until it receives a unit, and Task A can send a unit to the semaphore whenever it wishes to grant permission for Task B to run. If Tasks A and B are within the same job, this would be a straightforward use of a semaphore. But the two tasks are in different jobs, and this causes some complications.

Specifically, how do Tasks A and B access the same semaphore? For instance, Task A can create the semaphore and access it, but how can Task A provide Task B with a token for the semaphore? The trick is to use the object directory of the root job.

In the following explanation, each of the two tasks must perform half of a protocol. The process of creating and cataloging the semaphore is one half, and the process of looking up the semaphore is the other.

In order for this protocol to succeed, the programmers of the two tasks must agree on a name for the semaphore, and they must agree which task performs which half of the protocol. In this example, the semaphore is named permit_sem. And, because Task B must wait until Task A grants permission, Task A will create and catalog the semaphore, and Task B will look it up.

Task A performs the creating and cataloging as follows:

1) Task A creates a semaphore with no units by calling the CREATE$SEMAPHORE system call. This provides Task A with a token for the semaphore.

2) Task A calls the GET$TASK$TOKENS system call to obtain a token for the root job.

3) Task A calls the CATALOG$OBJECT system call to place a token for the semaphore in the object directory of the root job under the name permit_sem.

4) Task A continues processing, eventually becomes ready to grant permission, and sends a unit to permit_sem.

Task B performs the look-up protocol as follows:

1) Task B calls the GET$TASK$TOKENS system call to obtain a token for the root job.

2) Task B calls the LOOKUP$OBJECT system call to obtain a token for the object named permit_sem. If the name has not yet been cataloged, Task B waits until it is.

3) Task B calls the RECEIVE$UNITS system call to request a unit from the semaphore. If the unit is not available then Task A has not yet granted permission, and Task B waits. When a unit is available, Task A has granted permission, and Task B becomes ready.

There are several aspects of this technique that you should be aware of:

- In the example, the object directory technique was used to pass a semaphore. The same technique can be used to pass any type of iRMX 86 object.

- The semaphore was passed via the object directory of the root job. The root job's object directory is unique in that it is the only object directory to which all jobs in the system can gain access. This accessibility allows one job to "broadcast" an object to any job that knows the name under which the object is cataloged.

- The object directory of the root job must be large enough to accommodate the names of all the objects passed in this manner. If it is not, it will become full and the iRMX 86 Operating System will return an exception code when attempts are made to catalog additional objects.

- If you use this technique to pass many objects, you could have problems ensuring unique names. If name management becomes a problem, different sets of jobs can adopt the convention of using an object directory other than that of the root job. To accomplish this, one of the jobs catalogs itself in the root job's object directory under an agreed-upon name. The other jobs can then look up the cataloged job and use its object directory rather than that of the root job.

- In the example, the object-passing protocol was divided into two halves: the create-and-catalog half, and the look-up half. The protocol works correctly regardless of which half starts to run first.

PASSING OBJECTS THROUGH MAILBOXES

Another means of sending objects from one job to another is to use a mailbox. This is a two-step process in that the two jobs using the mailbox must first use the object directory technique to obtain mutual access to the mailbox, and then they use the mailbox to pass additional objects.

PASSING PARAMETER OBJECTS

One of the parameters of the CREATE$JOB system call is a parameter object. The purpose of this parameter is to allow a task in the parent job to pass an object to the newly created job. Once the tasks in the new job begin running, they can obtain a token for the parameter object by calling GET$TASK$TOKENS. This technique is illustrated in the following example:

Suppose that Task 1 in Job 1 is responsible for spawning a new job (Job 2). Suppose also that Task 1 maintains an array that is needed by Job 2. Task 1 can pass the array to Job 2 by putting the array into an iRMX 86 segment, and designating the segment as the parameter object in the CREATE$JOB system call. Then the tasks of Job 2 can call the GET$TASK$TOKENS system call to obtain a token for the segment.

In the foregoing example, the parameter object is a segment. However, you can use this technique to pass any kind of iRMX 86 object.

AVOID PASSING OBJECTS THROUGH SEGMENTS OR FIXED MEMORY LOCATIONS

In the current version of the iRMX 86 Operating System, tokens remain unchanged when objects are passed from job to job. However, Intel reserves the right to modify this rule. In other words, if you pass objects from one job to another and you want your software to be able to run on future releases of the iRMX 86 System, obey the following guidelines:

- Never pass a token from one job to another by placing the token in an iRMX 86 segment and then passing the segment.
- Never pass a token from one job to another by placing the token in any memory location that the two jobs both access.

COMPARISON OF OBJECT-PASSING TECHNIQUES

There are several guidelines to consider when deciding how to pass an object between jobs:

- If you are passing only one object from a parent job to a child job, use the parameter object when the parent creates the child.
- If you are passing only one object but not from parent to child, use the object directory technique. It is simpler than using a mailbox.
- If you need to pass more than one object at a time, you can use any of the following techniques:
    - Assign an order to the objects and send them to a mailbox where the receiving job can pick them up in order.
    - Give each of the objects a name and use an object directory.
    - Write a simple type manager that packs and unpacks a set of objects. Then pass the set of objects as one composite object.

***

CHAPTER 6 SIMPLIFYING CONFIGURATION DURING DEVELOPMENT

For your convenience, the configuration information found in this chapter has been added to the iRMX 86 CONFIGURATION GUIDE. For any information that you might need concerning the following topics, refer to the iRMX 86 CONFIGURATION GUIDE.

- Data segments
- Configuration
- Freezing locations of entry points
- The Interactive Configuration Utility (ICU)
- The LOC86 command
- Padding memory segments

***

CHAPTER 7 DEADLOCK AND DYNAMIC MEMORY ALLOCATION

This chapter is for anyone who writes tasks which dynamically allocate memory, send messages, create objects, or delete objects. In order to understand this chapter, you should be familiar with the following concepts:

- memory management in the iRMX 86 Operating System
- using either iRMX 86 semaphores or regions to obtain mutual exclusion

PURPOSE OF THIS CHAPTER

Memory deadlock is not difficult to diagnose or correct, but it is difficult to detect. Because memory deadlock generally occurs under unusual circumstances, it can lie dormant throughout development and testing, only to bite you when your back is turned. The purpose of this chapter is to provide you with some special techniques that can prevent memory deadlock.

HOW MEMORY ALLOCATION CAUSES DEADLOCK

The following example illustrates the concept of memory deadlock and shows the danger that iRMX 86 tasks can face when they cause memory to be allocated dynamically.

Suppose that the following circumstances exist for Task A and B which belong to the same job:

- Task A has lower priority than Task B.
- Each task wants two iRMX 86 segments of a given size, and each asks for the segments by calling the CREATE$SEGMENT system call repeatedly until both segments are acquired.
- The job's memory pool contains only enough memory to satisfy two of the requests.
- Task B is asleep and Task A is running.

Now suppose that the following events occur in the order listed:

1) Task A gets its first segment.

2) An interrupt occurs and Task B is awakened. Since Task B is of higher priority than Task A, Task B becomes the running task.

3) Task B gets its first segment.

The two tasks are now deadlocked. Task B remains running and continues to ask for its second segment. Not only are both of the tasks unable to progress, but Task B is consuming a great deal, perhaps all, of the processor time. At best, the system is seriously degraded.

This kind of memory allocation deadlock problem is particularly insidious because it quite likely would not occur during debugging. The reason for this is that the order of events is critical in this deadlock situation.

Note that the key event in the deadlock example is the awakening of Task B just after Task A invokes the first CREATE$SEGMENT system call, but just before Task A invokes the second CREATE$SEGMENT call. Because this critical sequence of events occurs only rarely, a "thoroughly debugged" system might, after a period of flawless performance, suddenly fail.

Such intermittent failures are costly to deal with once your product is in the field. Consequently, the most economical method for dealing with memory deadlock is to prevent it.

SYSTEM CALLS THAT CAN LEAD TO DEADLOCK

A task cannot cause memory deadlock unless it causes memory to be allocated dynamically. And the only means for a task to allocate memory is by using system calls. If your task uses any of the following system calls, you must take care to prevent deadlock:

- any system call that creates an object
- any system call belonging to a subsystem other than the Nucleus
- SEND$MESSAGE
- DELETE$JOB
- DELETE$EXTENSION

If a task uses none of the preceding system calls, it cannot deadlock as a result of memory allocation.

PREVENTING MEMORY DEADLOCK

Using any one of the following techniques, you can eliminate memory deadlock from your system:

- When a task receives an E$MEM condition code, the task should not endlessly repeat the system call that led to the code. Rather, it should repeat the call only a predetermined number of times. If the task still receives the E$MEM condition, it should delete all its unused objects, and try again. If the E$MEM code is still received, the task should sleep for a while and then reissue the system call.

- If you have designed your system so that a job cannot borrow memory from the pool of its parent, you can use an iRMX 86 semaphore or region to govern access to the memory pool. Then, when a task requires memory, it must first gain exclusive access to the job's memory pool. Only after obtaining this access may the task issue any of the system calls listed above.

  The task's behavior should then depend upon whether the system can satisfy all of the task's memory requirements:

  - If the system cannot satisfy all requirements, the task should delete any objects that were created and surrender the exclusive access. Then the task should again request exclusive access to the pool.

  - If, on the other hand, all requests are satisfied, the task should surrender exclusive access and begin using the objects.

  This technique prevents deadlock by returning unused memory to the memory pool, where it may be used by another task.

- If you have designed your system so that a job cannot borrow memory from the pool of its parent, prevent the tasks within the job from directly completing for the memory in the job's pool. You can do this by allowing no more than one task in each job to use the system calls listed earlier.

***

CHAPTER 8
GUIDELINES FOR STACK SIZES

This chapter is for three kinds of readers:

- Those who write tasks that create iRMX 86 jobs or tasks.
- Those who write interrupt handlers.
- Those who write tasks that are to be loaded by the Application Loader or tasks to be invoked by the Human Interface.

In order to understand all of this chapter, you must be familiar with the iRMX 86 Debugger, and you must know which system calls are provided by the various subsystems of the iRMX 86 Operating System. You also must know the difference between maskable and nonmaskable interrupts. Finally, if you are writing an interrupt handler, you must know what an interrupt handler is.

PURPOSE OF THIS CHAPTER

This chapter has three purposes. If your are writing a task that creates a job or another task, the purpose of this chapter is to help you compute the amount of stack that you must specify in the system call that performs the creation. If you are writing an interrupt handler, the purpose of this chapter is to inform you of stack size limitations to which you must adhere. If you are writing a task that is to be loaded by the Application Loader or invoked by the Human Interface, the purpose of this chapter is to show you how much stack to reserve during the linking and locating process.

STACK SIZE LIMITATION FOR INTERRUPT HANDLERS

Many tasks running in the iRMX 86 Operating System are subject to two kinds of interrupts -- maskable, and nonmaskable. When these interrupts occur, the associated interrupt handlers use the stack of the interrupted task. Consequently, you must know how much of your task's stack to reserve for these interrupt handlers.

The iRMX 86 Operating System assumes that all interrupt handlers, including those that you write, require no more than 128 (decimal) bytes of stack, even if a task is interrupted by both a maskable and a nonmaskable interrupt. If when writing an interrupt handler you fail to adhere to this limitation, you expose your system to the risk of stack overflow.

In order to stay within the 128 (decimal) byte limitation, you must restrict the number of local variables that the interrupt handler stores on the stack. For interrupt handlers serving maskable interrupts, you may use as many as 20 (decimal) bytes of stack for local variables. For handlers serving the nonmaskable interrupt, you may use no more than 10 (decimal) bytes. The balance of the 128 bytes is consumed by the SIGNAL$INTERRUPT system call, and by storing the registers on the stack.

For more information about interrupt handlers, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

STACK GUIDELINES FOR CREATING TASKS AND JOBS

Whenever you invoke a system call to create a task, you must specify the size of the task's stack. And, since every new job has an initial task that is created simultaneously with the job, you must also designate a stack size whenever you create a job.

When you specify a task's stack size, you should do so carefully. If you specify a number that is too small, your task might overflow its stack and write over information following the stack. This situation can cause your system to fail. On the other hand, if you specify a number that is too large, the excess memory will be wasted. So ideally, you should specify a stack size that is only slightly larger than what is actually required.

This chapter provides you with two techniques for estimating the size of your task's stack. One technique is arithmetic, and the other is empirical. For best results, you should start with the arithmetic technique and then use the empirical technique for tuning your original estimate.

STACK GUIDELINES FOR TASKS TO BE LOADED OR INVOKED

If you are creating a task that is to be loaded by the Application Loader or invoked by the Human Interface, you must specify the size of the task's stack during the linking or locating process. The arithmetic and empirical techniques in this manual will help you estimate the size of your task's stack.

ARITHMETIC TECHNIQUE

This technique provides you with a reasonable overestimate of your task's stack size. After you use this technique to obtain a first approximation, you may be able to save several hundred bytes of memory by using the empirical technique described later in this chapter.

The arithmetic technique is based on the fact that there are at most three factors affecting a task's stack. These factors are:

- interrupts
- iRMX 86 system calls
- requirements of the task's code (For example, the stack used to pass parameters to procedures or to hold local variables in reentrant procedures.)

You can estimate the size of a task's stack by summing the amount of memory needed to accommodate these factors. The following sections explain how to compute the stack requirements for the first three factors.

STACK REQUIREMENTS FOR INTERRUPTS

Whenever an interrupt occurs while your task is running, the interrupt handler uses your task's stack while servicing the interrupt. Consequently, you must ensure that your task's stack is large enough to accommodate the needs of two interrupt handlers -- one for maskable interrupts, and one for nonmaskable interrupts. All interrupt handlers used with the iRMX 86 Operating system are designed to to ensure that, even if two interrupts occur (one maskable, one not), no more than 128 (decimal) bytes of stack are required by the interrupt handlers.

STACK REQUIREMENTS FOR SYSTEM CALLS

When your task invokes an iRMX 86 system call, the processing associated with the call uses some of your task's stack. The amount of stack required depends upon which system calls you use.

Table 8-1 tells you how many bytes of stack your task must have to support various system calls. To find out how much stack you must allocate for system calls, compile a list of all the system calls that your task uses. Scan Table 8-1 to find which of your system calls requires the most stack. By allocating enough stack to satisfy the requirements of the most demanding system call, you can satisfy the requirements of all system calls used by your task.

Table 8-1. Stack Requirements For System Calls

System Calls	Bytes (Decimal)
S$SEND$COMMAND CGETINPUT$PATHNAME C$GET$OUTPUT$PATHNAME CGETINPUT$CONNECTION C$GET$OUTPUT$CONNECTION	800
ALL OTHER HUMAN INTERFACE SYSTEM CALLS	600
S$LOAD$IO$JOB	440
A$LOAD$IO$JOB A$LOAD S$OVERLAY	400
EXTENDED I/O SYSTEM CALLS	400
BASIC I/O SYSTEM CALLS	300
CREATE$JOB DELETE$EXTENSION DELETE$JOB DELETE$TASK FORCE$DELETE RESET$INTERRUPT	225
ALL OTHER NUCLEUS CALLS	125

COMPUTING THE SIZE OF THE ENTIRE STACK

To compute the size of the entire stack, add the following three numbers:

- the number of bytes required for interrupts (128 decimal bytes)
- the number of bytes required for system calls
- the amount of stack required by the task's code segment

You can use the sum of these three numbers as a reasonable estimate of your task's stack requirements. If you desire more accuracy, use the sum as a starting point for the empirical fine tuning described later in this chapter.

EMPIRICAL TECHNIQUE

This technique starts with an overly large stack and uses the iRMX 86 Debugger to determine how much of the stack is unused. Once you have found out how much stack is unused, you can modify your task- and job-creation system calls to create smaller stacks.

The cornerstone of this technique is the iRMX 86 Debugger. In order to use the Debugger, you must include it when you configure your application system. Information on how to do this is provided in the iRMX 86 CONFIGURATION GUIDE.

The Inspect Task command of the Debugger provides a display that includes the number of bytes of stack that have not been used since the task was created. If you let your task run a sufficient length of time, you can use the Inspect Task command to find out how much excess memory is allocated to your task's stack. Then you can adjust the stack-size parameter of the system call to reserve less stack.

The only judgment you must exercise when using this technique is deciding how long to let your task run before obtaining your final measurement. If you do not let the task run long enough, it might not encounter the most demanding combination of interrupts and system calls. This could cause you to underestimate your task's stack requirement and could, consequently, lead to a stack overflow in your final system.

Underestimation of stack size is a risk inherent in this technique. For example, your task might be written so as to use its peak demand for stack only once every two months. Yet you probably don't want to let your system run for two months just to save several hundred bytes of memory. You can avoid such excessive trial runs by padding the results of shorter runs. For instance, you might run your task for 24 hours and then add 200 (decimal) bytes to the maximum stack size. This padding reduces the probability of overflowing your task's stack in your final system.

***

INDEX

Primary references are underscored.

***

iRMX™ 86 TERMINAL HANDLER REFERENCE MANUAL

CONTENTS

***

CHAPTER 1
OVERVIEW OF THE TERMINAL HANDLER

The Terminal Handler supports real-time, asynchronous I/O between an operator's terminal and tasks running under the iRMX 86 Nucleus. It is intended for use in applications which require only limited I/O through a terminal, and it generally is used in applications that do not include the iRMX 86 I/O System. The features of the Terminal Handler include the following:

- Line editing capabilities.
- Keystroke control over output, including output suspension and resumption, and deletion of data being sent by tasks to the terminal.
- Echoing of characters as they are entered into the Terminal Handler's line buffer.

An output-only version of the Terminal Handler is available for use in applications in which tasks send output to a terminal but do not receive input from the terminal.

NOTE

The terminal handler is intended primarily to support character-by-character input from a terminal, rather than computer-to-computer input.

ORGANIZATION OF THIS MANUAL

This manual consists of four chapters:

- Chapter 1 --Overview of the Terminal Handler

  This chapter discusses the purpose of the Terminal Handler and introduces some of the features.

- Chapter 2 -- Using a Terminal with the iRMX 86 Operating System

  This chapter provides information that the operator needs in order to use a terminal with the iRMX 86 Operating System.

- Chapter 3 -- Programming Considerations

  This chapter contains the information that a programmer needs to write tasks that send data to, or receive data from, the terminal.

- Chapter 4 -- Configuration

  This chapter identifies and describes the configurable features, characteristics, and identifiers of the Terminal Handler.

***

CHAPTER 2
USING A TERMINAL WITH THE iRMX™ 86 OPERATING SYSTEM

When you are using a terminal with the iRMX 86 Operating System, you must limit the maximum priority of your tasks or they could interfere with the proper functioning of your terminal. High priority processor-bound tasks can cause the Terminal Handler to drop input characters.

While using a terminal that is under control of the Terminal Handler, an operator either reads an output message from the terminal's display or enters characters by striking keys on the terminal's keyboard. Normal input characters are those destined for input messages that are sent to tasks. Special input characters direct the Terminal Handler to take special actions. The special characters are RUBOUT, Carriage Return, Line Feed, ESCape, control-C, control-O, control-Q, control-R, control-S, control-X, and control-Z. The output-only version of the Terminal Handler does not support any of the special characters. In the remainder of this chapter, the handling of these two types is discussed, and the significance of each of the special characters is explained.

NOTE

This chapter contains several references to mailboxes and request messages used by tasks to communicate with the terminal. If you are puzzled by such a reference, look in Chapter 3 for an explanation.

HOW NORMAL CHARACTERS ARE HANDLED

The destination of a normal character, when entered, depends on whether there is an input request message at the Terminal Handler's input request mailbox. If there is an input request message, the character is echoed to the terminal's display and goes into the input request message. If there is not an input request message, the character is deleted.

HOW SPECIAL CHARACTERS ARE HANDLED

Table 2-1 lists the special characters and summarizes the effects of each of them. The following text comprises complete descriptions of the effects of the special characters. In these descriptions, there are several references to "the current line." The current line consists of the data, with editing, that has been entered since the last end-of-file character.

Table 2-1. Special Character Summary

Special Character	Effect
RUBOUT	Deletes previously entered character.
Carriage Return	Signals end of line.
Line Feed	Signals end of line.
ESCape	Signals end of line.
control-C	Calls an application program.
control-O	Kills or restarts output.
control-Q	Resumes suspended output.
control-R	Displays current line with editing.
control-S	Suspends output.
control-X	Deletes the current line.
control-Z	Sends empty message.

The following descriptions concern the special characters needed when entering data at the terminal. Most of these characters are for line-editing. Each description is divided into two parts: internal effects and external effects. The difference is that internal effects are those that are not directly visible, whereas external effects are immediately shown on the terminal's display.

RUBBING OUT A PREVIOUSLY-TYPED CHARACTER (RUBOUT)

Internal Effects: Causes the most recently entered but not yet deleted character to be deleted from the current line. If the current line is empty, there is no internal effect.

External Effects: If the current line is empty, the BEL character (07H) is sent to the terminal. Otherwise, the character is "rubbed out" in accordance with one of two available rubout modes. See Chapter 4 for a description of rubout modes.

DISPLAYING THE CURRENT LINE (CONTROL-R)

Internal Effects: None.

External Effects: Sends a carriage return and line feed to the terminal, followed by the current line. If the current line is empty, the previous line is sent to the display, where it can be line-edited and submitted as a new input message.

DELETING THE CURRENT LINE (CONTROL-X)

Internal Effects: Empties the current line.

External Effects: Causes the sequence (#, Carriage Return, Line Feed) to be sent to the terminal.

SENDING AN EMPTY MESSAGE (CONTROL-Z)

Internal Effects: Puts a zero in the ACTUAL field of the input request message currently being processed. The message is then sent to the appropriate response mailbox.

External Effects: None.

SIGNALLING THE END OF A LINE OF INPUT (CARRIAGE RETURN, LINE FEED, OR ESCAPE)

Internal Effects: Puts either the ASCII end-of-transmission character (OAH in the case of Carriage Return or Line Feed) or the ESCape character (1BH) in the current line. Each of these characters signals the end of a message, so the input request message currently being constructed is sent to the appropriate response mailbox.

External Effects: If the end-of-line indicator is either Carriage Return or Line Feed, both Carriage Return and Line Feed are sent to the terminal. If the indicator is ESCape, however, there is no effect on the display.

OUTPUT CONTROL

Output request messages that are sent to output mailboxes can be processed in one of three ways:

- They can be outputted as described later in Chapter 3.
- They can be queued at the output mailbox where they remain until an operator at the terminal takes action to permit processing of the messages.
- They can be discarded.

In the descriptions that follow, these methods of dealing with output requests are called the normal mode, the queueing mode, and the suppression mode, respectively. Initially, output is in the normal mode.

SUSPENDING OUTPUT (CONTROL-S)

Puts output in the queueing mode.

RESUMING OUTPUT (CONTROL-Q)

Negates the effects of control-S by allowing the display of output requests that are sent to the output mailbox. The output that has been suppressed is displayed (very quickly) in the order in which it would have been displayed earlier if the control-S had not been pressed. If you are overwhelmed by this output, you can stop it again by pressing control-S.

DELETING OR RESTARTING OUTPUT (CONTROL-O)

If output is in the normal mode, control-O puts it in the suppression mode. If output is in the suppression mode, control-O restores it to the normal mode. If output is in the queueing mode, control-O has no effect. Internally, the request messages that tasks send while output is being suppressed are returned to those tasks just as if the output had not been suppressed.

PROGRAM CONTROL

The remaining control character affects system behavior.

CALLING A USER-WRITTEN PROCEDURE MANUALLY (CONTROL-C)

Control-C invokes a parameter-less, user-written procedure named RQ$ABORT$AP. This procedure, which must be compiled under the COMPACT control, can perform any actions that suit the application. Often, as its name suggests, RQ$ABORT$AP aborts an application. If it is written by the user (and it need not be), RQ$ABORT$AP is not required to have a RETURN statement.

Control-C also causes the effects produced by control-Z; that is, it returns the current input request message with its ACTUAL field set to zero. This is the case even if the application system does not contain an RQ$ABORT$AP procedure.

***

CHAPTER 3
PROGRAMMING CONSIDERATIONS

The iRMX 86 Terminal Handler supports terminal input and output by providing mailbox interfaces. Figure 3-1 shows the use of these mailboxes. In the figure, an arrow pointing from a task to a mailbox represents an RQ$SEND$MESSAGE system call. An arrow pointing from a mailbox to a task indicates an RQ$RECEIVE$MESSAGE system call.

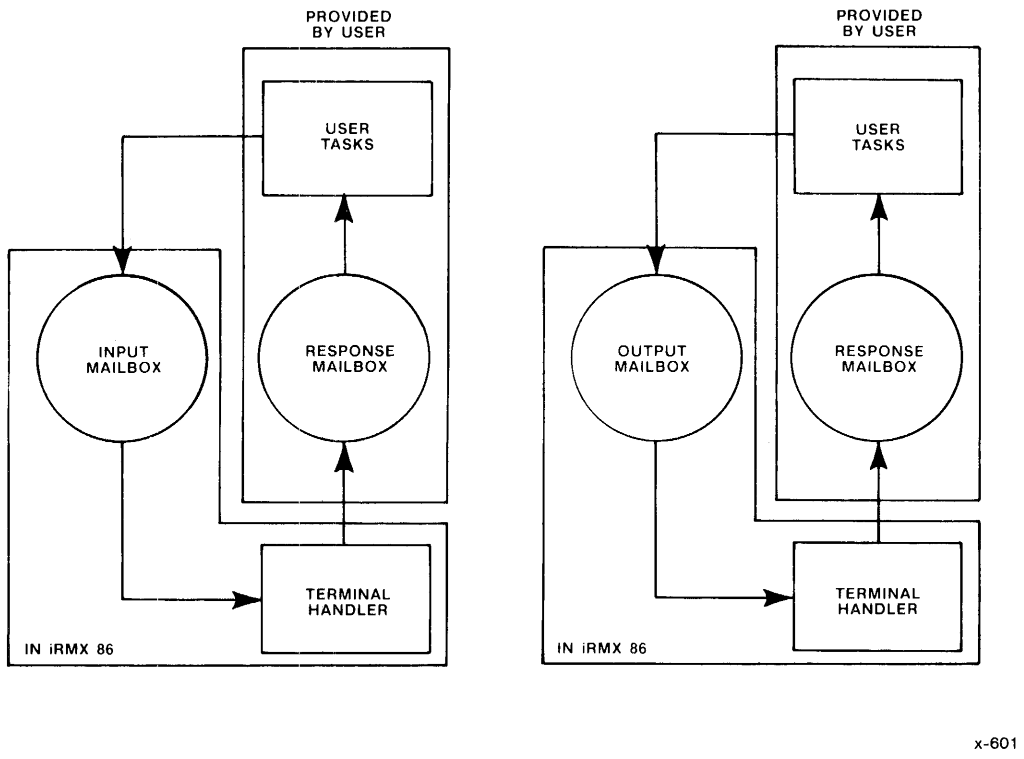


Figure 3-1. Input And Output Mailbox Interfaces

The protocol that tasks observe is much the same for input and output. In each case, the task initiates I/O by sending a request message to a mailbox. An input request mailbox (default name RQTHNORMIN) and an output request mailbox (default name RQTHNORMOUT) are provided. These mailboxes are cataloged in the root job directory. In the case of multiple terminals, one input and one output mailbox will be cataloged for each Terminal Handler. (See Chapter 4 for more information about multiple versions of the Terminal Handler.) Figure 3-2 illustrates the protocol for finding the root job token and for obtaining the input and output mailbox tokens.

```
/*************************************************************************
*  This example illustrates the protocol for finding the root job token *
*  and for obtaining the input and output mailbox tokens.               *
*************************************************************************

DECLARE rtjb$token                      WORD;
DECLARE root$job                        LITERALLY '3';
DECLARE status                          WORD;

DECLARE input$mbx$token                 WORD;

DECLARE wait$forever                    LITERALLY '0FFFFH';


/*By setting the input parameter to three, the GET$TASK$TOKEN primitive
  will return the root job's TOKEN.*/

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

/*The following LOOKUP$OBJECT primitives use the default mailbox names.*/

input$mbx$token = RQ$LOOKUP$OBJECT      (rtjb$token,
                                        @(10, 'RQTHNORMIN'),
                                        wait$forever,
                                        @status);

output$mbx$token = RQ$LOOKUP$OBJECT     (rtjb$token,
                                        @(11, 'RQTHNORMOUT'),
                                        wait$forever,
                                        @status);
```

Figure 3-2. Protocol For Obtaining Root Job And Mailbox Tokens

Refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for more information concerning the individual primitives used in the previous example. When a task sends a message to the Terminal Handler mailbox, the Terminal Handler processes the request and then sends a response message back to the requesting task. The task waits at a response mailbox for the message. Thus, whether a task does input or output, it first sends and then receives. The full details of the input and output protocols are described later in this chapter. Output is discussed first because it is somewhat easier to understand.

For both input and output, a task sends a message segment to the Terminal Handler. The format of a request message is depicted in Figure 3-2. The numbers in that figure are offsets, in bytes, from the beginning of the segment. The field names have different meanings for input and for output. For both input and output, the first four fields are WORD values. The MESSAGE CONTENT field can be up to 132 bytes in length for input and up to 65527 bytes in length for output.

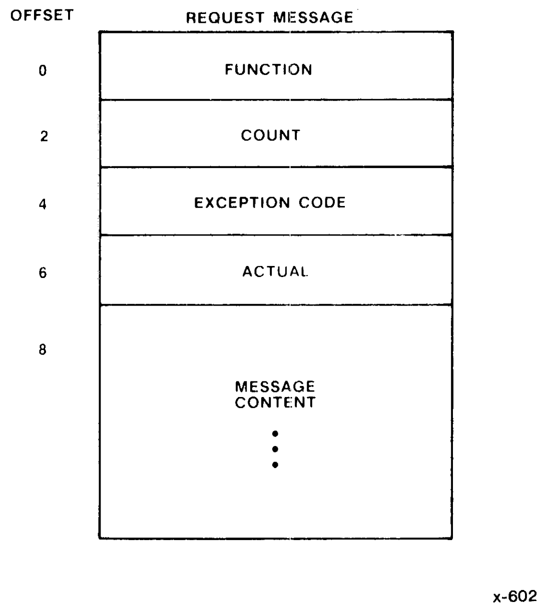


Figure 3-3. Request Message Format

In the following discussions, the names F$WRITE and F$READ are literal names for the particular WORD values 5 and 1, respectively.

OUTPUT

The first thing a task does when transmitting output is prepare an output request message. The task must fill in the following fields prior to sending the message:

FUNCTION --- F$WRITE.

COUNT --- the number of bytes (not to exceed 65527) in the MESSAGE CONTENT field.

MESSAGE CONTENT --- the bytes that are to be output.

Having prepared the message segment, the task must send it to the output request mailbox. Messages sent to this mailbox are processed in a first-in-first-out manner. Processing a message involves sending the characters in the MESSAGE CONTENT field to the terminal until a total of COUNT characters have been sent. There is one exception; when the Terminal Handler encounters the end-of-transmission character (OAH), it sends a Carriage Return and a Line Feed to the terminal.

When sending the output request message, the task specifies a user-supplied response mailbox. If no response mailbox is specified, the Terminal Handler will delete the segment that contained the message. But note, if the the system call DISABLE$DELETION was used by the application engineer to make the segment containing the output message immune to ordinary deletion, the Nucleus will put the Terminal Handler into the asleep state because the Terminal Handler cannot execute deletion of the segment. This situation effectively eliminates the Terminal Handler as a functioning task.

In addition to transmitting the message to the terminal, the Terminal Handler fills in the remaining fields in the output request message. The requesting task can wait indefinitely at the response mailbox (that is, it can call the RQ$RECEIVE$MESSAGE system call with a time limit of OFFFFH) immediately after sending the output request. By observing this protocol, the task can learn of the success or failure of the output attempt. The fields that provide this information are the following:

- EXCEPTION CODE --- the encoded result of the output operation:
  - E$OK --- the operation was successful.
  - E$PARAM --- the FUNCTION field in the message did not contain F$WRITE.
  - E$BOUNDS --- the COUNT field in the message is too big for the segment, that is, COUNT + 8 is greater than the length of the segment containing the message.
- ACTUAL --- the actual number of bytes output.

In summary, the protocol observed by tasks doing output is as follows:

- Prepare the output request message segment, filling in the FUNCTION, COUNT, and MESSAGE CONTENT fields.
- Send the segment, via the RQ$SEND$MESSAGE system call, to the output request mailbox. It is advisable, but not necessary, to specify a response mailbox in the system call.
- Wait indefinitely, via the RQ$RECEIVE$MESSAGE system call, at the response mailbox. When received, the message contains the results of the transmission in the EXCEPTION CODE and ACTUAL fields.

INPUT

The protocol for obtaining input is much the same as that for outputting. A message is prepared and sent to a request mailbox; then, after the data has been input, the message is received at a response mailbox. There is a significant difference, however, between input and output protocols. Because the input is contained in the message segment at the response mailbox, it is necessary to designate a response mailbox and then wait there.

CAUTION

When multiple tasks use the same mailbox for input from the terminal, it is possible for a task to get input that is intended for another task.

A task needing input first prepares an input request message. It must fill in the FUNCTION and COUNT fields prior to sending its request. The FUNCTION field must contain F$READ. The COUNT field reflects the maximum possible number of input characters in the input message. The value of COUNT must not exceed 132; moreover, COUNT + 8 must not exceed the length of the input request message segment.

When sending the input request message, the task must specify the response mailbox in its call to RQ$SEND$MESSAGE. The Terminal Handler obtains characters from the terminal and places them in the MESSAGE CONTENT field. The message is terminated by an end-of-line character (Carriage Return, Line Feed, or ESCape). The lone exception is when the end-of-line character has been "normalized" by being preceded by a control-P; then the end-of-line character is treated as a normal character.

NOTE

If more than COUNT characters are entered prior to the end-of-line character, the extra characters are ignored, and the control-G character is activated. This character usually causes the terminal to emit a beep tone.

After the message is complete, the Terminal Handler fills in the EXCEPTION CODE and ACTUAL fields as follows:

- EXCEPTION CODE --- the encoded result of the input operation, which is one of the following:

  - E$OK --- the operation was successful.

  - E$PARAM --- either the FUNCTION field in the message did not contain F$READ or the COUNT field was greater than 132.

  - E$BOUNDS --- COUNT + 8 is greater than the length of the message segment.

- ACTUAL --- the number of bytes actually entered and placed in the MESSAGE CONTENT field.

The requesting task must wait indefinitely (that is, it must make a RQ$RECEIVE$MESSAGE system call with a time limit of OFFFFH) at the designated response mailbox immediately after sending the input request.

In summary, the input protocol is as follows:

- Prepare the input request message segment, filling in the FUNCTION and COUNT fields.

- Send the segment, via the RQ$SEND$MESSAGE system call, to the input request mailbox. In the call, specify a response mailbox.

- Wait indefinitely, via the RQ$RECEIVE$MESSAGE system call, at the response mailbox. When received, the message segment will contain the results of the input operation in the MESSAGE CONTENT, EXCEPTION CODE, and ACTUAL fields.

***

CHAPTER 4 CONFIGURATION

The Terminal Handler is a configurable layer of the Operating System. It contains several options that you can adjust to meet your specific needs. To make configuration choices, Intel provides three kinds of information:

- A list of configurable options.
- Detailed information about the options.
- Procedures to allow you to specify your choices.

The balance of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the iRMX 86 CONFIGURATION GUIDE.

CONFIGURABLE OPTIONS

Some Terminal Handler features, characteristics, and identifiers are configurable. Configurability is important for applications with unusual characteristics, such as a component hardware environment or multiple terminal handlers. The following sections describe the configurable options available on the Terminal Handler.

SELECTING A VERSION OF THE TERMINAL HANDLER

The iRMX 86 Terminal Handler is available in two different versions:

- Input and Output
- Output-only

The input and output version allows you to enter characters at the terminal as well as receive data. The output-only version is useful in applications in which tasks send output to a terminal but do not receive input from the terminal.

BAUD RATE

You can set the baud rate for the Terminal Handler to any of the following values:

110
150
300
600
1200
2400
4800
9600
19200

The default baud rate for the Terminal Handler is 9600.

Baud Count

The baud count provides a way to calculate internal timer values given the clock input frequency. If your system's programmable interval timer (PIT) has a clock input frequency other than 1.2288 megahertz, you must set the baud count. The default value for the baud count is 4.

RUBOUT MODE AND BLANKING CHARACTER

As previously mentioned, there are two ways to rubout a character:

- Copying mode
- Blanking mode

In the copying mode, the character being deleted from the current line is re-echoed to the display. For example, entering "CAT" and then striking RUBOUT three times results in the display "CATTAC".

In the blanking mode, the deleted character is replaced on the CRT screen with the blanking character. For example, entering "CAT" and then striking RUBOUT three times deletes all three characters from the display.

The copy mode is the default mode. The default blanking character for the blanking mode is a space (20H).

CONFIGURATION

USART

The USART is a device that, depending upon the application, can be used either to convert serial data to parallel data or to convert parallel data to serial data. The Terminal Handler requires a 8251A USART as a terminal controller. You can specify

- The port address of the USART. The default value for the port address is 0D8H
- The interval between the port addresses for the USART.
- The number of bits of valid data per character that can be sent from the USART. The default value for the number of bits is 7.

PIT

The Terminal Handler requires a PIT as an input to the USART. You can specify the following information about the programmable interval timer (PIT):

- The port address of the PIT. The default value for the port address is 0D0H.
- The interval between the port addresses for the PIT.
- The number of the PIT counter connected to the USART clock input. The default value is 2.

MAILBOX NAMES

You can change the default names of both the input mailbox (RQTHNOTMIN) and the output mailbox (RQTHNORMOUT). The new names must not be over 12 alphanumeric characters in length.

INTERRUPT LEVELS

You can specify the interrupt levels used by the Terminal Handler for input and output. You choose interrupt levels by selecting a value that corresponds to a particular interrupt value. The default value for the input interrupt level is 68H and the default value for the output interrupt level is 78H.

CREATING MULTIPLE VERSIONS OF THE TERMINAL HANDLER

Your iRMX 86 system can contain multiple version of the Terminal Handler. This may be desirable if, for example, you have two tasks that use the Terminal Handler and you want to communicate with these tasks from separate terminals. In order to create multiple versions of the Terminal Handler, you must obey the following rules:

- Each Terminal Handler must use different input and output mailbox names.
- Each Terminal Handler must use a unique USART.
- Each Terminal Handler must use different interrupt levels.
- The code for the Terminal Handlers must be located in different, non-overlapping areas; each Terminal Handler must have its own data area.
- Each Terminal Handler must have a separate job.

Refer to the iRMX 86 CONFIGURATION GUIDE for detailed information about the previously described rules. If you adhere to these rules, you can create multiple versions of the Terminal Handler in your application system.

***

INDEX

Primary references are underscored.

***

iRMX™ 86 DEBUGGER REFERENCE MANUAL

CONTENTS

***

CHAPTER 1 INTRODUCTION

The development of almost every software application requires debugging. To aid in the development of iRMX 86 based systems, Intel provides various debugging tools.

OVERVIEW OF iRMX™ 86 OPERATING SYSTEM DEBUGGING TOOLS

This section will give an overview of some of the debugging tools available from Intel for iRMX 86 based systems

iRMX™ 86 DEBUGGER

The first tool available for system debugging is the iRMX 86 Debugger. The Debugger's essential power is in its ability to allow a software engineer to dynamically examine the data structures handled by the iRMX 86 operating system. For example, the Debugger can show which tasks are waiting at a particular mailbox while the application program is running. This capability permits you to easily debug a multitasking operation.

The Debugger supplies its own Terminal Handler, which includes all of the capabilities described in the iRMX 86 TERMINAL HANDLER REFERENCE MANUAL. Your application software can make use of the Debugger's Terminal Handler, or you can include a separate version (or versions) of the Terminal Handler in your system configuration for application use. Refer to the iRMX 86 CONFIGURATION GUIDE for further configuration information.

SYSTEM DEBUG MONITORS

The second tool available to the programmer is the Intel series of monitors. In this group are the iRMX 86 System Debug Monitor (SDB), the iSDM 86 monitor, and the iSBC 286 monitor. All of these system debug monitors can, among many other functions, single step instruction code, set execution and memory breakpoints, display memory in various formats (such as ASCII), perform I/O read and write operations, and move, search, and compare blocks of memory. The iRMX System Debugger Monitor (SDB) extends the use of the monitors so that you can directly examine Operating System data structures. For more information on the monitors, consult the following manuals: iSDM 86 SYSTEM DEBUG MONITOR REFERENCE MANUAL; the iSDM 286 SYSTEM DEBUG MONITOR REFERENCE MANUAL; or the iRMX 86 RELEASE 6 SYSTEM DEBUG REFERENCE MANUAL.

IN-CIRCUIT EMULATORS

The third debugging tool provided by Intel is the In-Circuit Emulator (ICE). The ICE lets you get closer to the system hardware by permitting you to examine the state of input pins and output ports, to set breakpoints, to look at the most recent 80 to 150 assembly language instructions, and to use the memory of your Intel Microprocessor Development System as if that memory were on your prototype board. For more information on the capabilities of the ICE, consult the ICE-86/ICE-88 MICROSYSTEMS IN-CIRCUIT EMULATOR OPERATION INSTRUCTIONS FOR ISIS-II USERS MANUAL.

CRASH ANALYZER

The fourth tool available for debugging iRMX 86 based systems is the Crash Analyzer. The Crash Analyzer is an utility used to dump the contents of memory into a file while formatting that information for display or printing. The file produced from the memory dump will allow you to see the state of every object at the time of the dump. For example, the programmer can see the size of memory segments when the dump occurred. For more information on the Crash Analyzer see the iRMX 86 CRASH ANALYZER REFERENCE MANUAL.

iRMX™ 86 DEBUGGER IMPLEMENTATION ON iAPX 186/286 CPUs

One of the advantages of the iRMX 86 Operating System is that it can be implemented using one of several Intel microprocessors. As a result, the use of the iRMX 86 Debugger does not change even though the microprocessor running the operating system maybe the iAPX 86, iAPX 186, iAPX 88, iAPX 188, or the iAPX 286 CPUs. This occurs because the Debugger acts on those features, such as registers, which the iAPX 86 and iAPX 186/286/88/188 microprocessors have in common. Thus, the iRMX 86 Debugger will appear to see an iAPX 86 CPU although another microprocessor may be physically running the system. (In the iAPX 286 microprocessor this is achieved by using the Real Address Mode.)

The same principle of compatability mentioned previously applies for the Numeric Processor Extension (NPX) used by the particular microprocessors. The iRMX 86 Debugger will see only those registers which the 8087 NPX has in common with the other Numeric Processor Extensions (e.g. the iAPX 286/20 processor).

OVERVIEW OF THE CAPABILITIES OF THE iRMX™ 86 DEBUGGER

The iRMX 86 Debugger enables you to do the following:

- Use the Debugger as a task, job, or system exception handler.
- View iRMX 86 object lists, including the lists of jobs, tasks, ready tasks, suspended tasks, asleep tasks, task queues at exchanges, object queues at mailboxes, exchanges, and iRMX 86 segments.
- Inspect jobs, tasks, exchanges, segments composites, and extensions.
- Examine and/or alter the contents of absolute memory locations.
- Set, change, view, and delete breakpoints.
- View the list of tasks that have incurred breakpoints and remove tasks from it.
- Declare a task to be the breakpoint task.
- Examine and/or alter the breakpoint task's register values.
- Set, change, view, and delete special variables for debugging.
- Deactivate the Debugger.

INVOKING THE DEBUGGER

You can invoke the Debugger from your iRMX 86 terminal by entering:

```
control-D
```

The Debugger responds with its sign-on message:

```
iRMX 86 DEBUGGER <version no.>
Copyright <year> Intel Corporation
*
```

The asterisk is the prompt character for the Debugger. It indicates that the Debugger is ready to accept additional input from the terminal.

In addition to the functions the Debugger can perform when it has been invoked, there are two services it can perform at any time, even when not invoked. First, if a task encounters a breakpoint, the Debugger responds as described in Chapter 4.

Second, if a task has the Debugger as its exception handler and the task causes an exceptional condition, then the Debugger displays a message to that effect at the terminal. A task can get the Debugger as its exception handler in one of the following ways:

- By using the SET$EXCEPTION$HANDLER system call.

- By acquiring the Debugger as the default exception handler. This is done at configuration time. Refer to the iRMX 86 CONFIGURATION GUIDE for a description of the this process.

- By having the Debugger declared as the exception handler when the task is created with CREATE$JOB or CREATE$TASK. An example of code setting up one of these calls is the following:

```
RQ$DEBUGGER$EX: PROCEDURE (EX$CODE, PARAM$NO, RESERVED,
                           NDP$STATUS, DUMMY$IF$COMPACT) EXTERNAL;
     DECLARE
          EX$CODE              WORD,
          PARAM$NO             BYTE,
          RESERVED             WORD,
          NDP$STATUS           WORD,
          DUMMY$IF$COMPACT     WORD;
END RQ$DEBUGGER$EX;

     DECLARE EXCEPT$BLOCK STRUCTURE (
          EXCEPT$PROC          POINTER,
          EXCEPT$MODE          BYTE);
          .
          .
          .

EXCEPT$BLOCK.EXCEPT$PROC = @RQ$DEBUGGER$EX;
EXCEPT$BLOCK.EXCEPT$MODE = ZERO$ONE$TWO$OR$THREE;
          .
          .
          .

RQ$CREATE$JOB(...,@EXCEPT$BLOCK,...);
  or
RQ$CREATE$TASK(...,@EXCEPT$BLOCK,...);
```

For this code to work, the task code must be linked to the CROOT.LIB library that is supplied with the Nucleus. The DUMMY$IF$COMPACT parameter in the RQ$DEBUGGER$EX declaration is a dummy parameter that you must include if your task is compiled using the PL/M-86 COMPACT.

***

CHAPTER 2 SPECIAL CHARACTERS

In addition to the Debugger commands listed in Chapter 4, the Debugger recognizes several special characters. This chapter lists these characters and describes their functions.

END-OF-LINE CHARACTERS

The Debugger obtains input one line at a time from its Terminal Handler. The end-of-line characters separate individual input lines. The Debugger recognizes three end-of-line characters. They are:

Carriage Return
Line Feed
ESCape

Both Carriage Return and Line Feed send the current input line to the Debugger for processing. ESCape causes the Debugger to discard the current input line and display a prompt.

CONTROL-S

The Debugger generates display at the iRMX 86 terminal by sending output messages to its Terminal Handler. Application tasks can also send messages to the same terminal. To suppress output from application tasks during a debugging session, type control-S. The Debugger then stores the output from application tasks until you type control-Q. If you do not enter control-S, any output from tasks is interspersed with output from the Debugger. Control-S has no effect on output from the Debugger.

CONTROL-Q

Control-Q negates the effect of a previously entered control-S character. To resume the output from tasks, type control-Q. Control-Q also causes the Debugger to display all output that was suppressed by control-S. Control-Q has no effect on output from the Debugger.

CONTROL-O

Certain Debugger command responses are lengthy and can roll off the screen. To freeze the top part of such a display before it disappears, enter control-O. This discards all output including Debugger prompts until you enter another control-O. The discarded output cannot be retrieved.

CONTROL-D

Occasionally you will want to terminate a Debugger memory command function response before it is finished. For example, if you asked for a display of memory locations 0000H to 0FFFFH, it would be natural to change your mind. To abort the display and regain the Debugger prompt, enter control-D.

Note that control-O affects the display only, whereas control-D stops the function entirely.

***

CHAPTER 3 COMMAND SYNTAX

When using the iRMX 86 Debugger, you sit at a terminal and type commands. This chapter describes the syntactical standards for commands to the Debugger, and it introduces notational conventions that are used throughout this manual.

CONVENTIONS

The first one or two characters of a command constitute a key sequence for the command:

- Most Debugger commands are specified by one or two letters. The key letters or pairs of letters are BL, BT, D, DB, G, I, L, M, N, Q, R, V, and Z.

- In a few cases, a command is specified by beginning the command with a name. A name, for the Debugger, must consist of a period followed by a variable name.

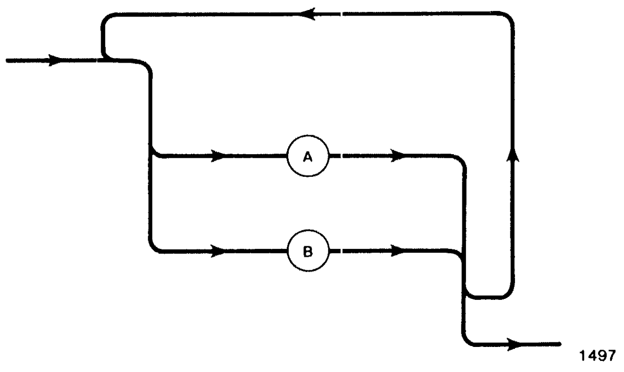
After the key initial sequence, a command may be followed by one or more parameters or additional specifiers. Blanks are used as delimiters between elements of a command; they are mandatory except as follows:

- Immediately after a command key that is not a name.

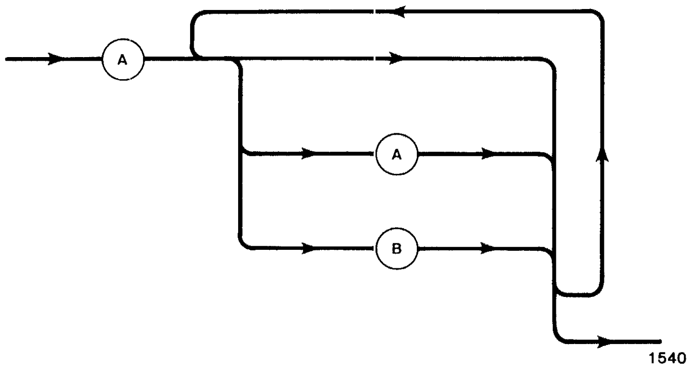
- Between a letter or digit and a non-letter, non-digit. The legal characters of the latter type are the following: ; @ = / : ( ) * + - ,

PICTORIAL REPRESENTATION OF SYNTAX

In this manual, a schematic device illustrates the syntax of commands. The schematic consists of what looks like an aerial view of a model railroad setup, with syntactic entities scattered along the track. Imagine that a train enters the system at the upper left, drives around as much as it can or wants to (sharp turns and backing up are not allowed), and finally departs at the lower right. The command it generates in so doing consists, in order, of the syntactic entities that it encounters on its journey. For example, a string of A's and B's, in any order, would be depicted as:



If such a string has to begin with an A, the schematic could be drawn as:



In the second drawing, it is necessary to represent the letter A twice because A is playing two roles. It is the first symbol (necessarily) and it is a symbol that may (optionally) be used after the first symbol. Note that a train could avoid the second A but cannot avoid the first A. The arrows are not necessary and henceforth are omitted.

SPECIAL SYMBOLS FOR THE DEBUGGER

The entities that will be used in the remainder of this manual, as A and B were used in the previous paragraph, are the following:

- CONSTANT. Constants are always hexadecimal. Unlike such constants in PL/M-86, they do not require an H as the last character. H's may be used if desired. Leading zeroes are not necessary unless they help to distinguish between constants and other things. For example, AH is a register in the iAPX 86, but 0AH is a constant.

- NAME. A name is a period followed by up to 11 characters, the first of which must be alphabetic. The other characters can be alphabetic, numeric, question marks (?), or dollar signs ($).

  Examples:

  ```
  .task

  .mailbox$7
  ```

- ITEM. An item is either an expression or one of the segment registers of the given CPU. The values of items are used variously as tokens and as offsets in Debugger commands. Graphically, an item is defined in Figure 3-2.

- EXPRESSION. As in algebra, an expression is either a term or is the result of adding and subtracting terms. Also as in algebra, a term is a product; each factor in the product is either a constant, a name, a parenthetical expression, or one of the registers AX, BX, CX, DX, DS, ES, SS, CS, IP, FL, SI, DI, BP, and SP. Graphically, term and expression are shown in Figure 3-1:

NOTE

If the computed value of an expression is too large to fit into four hexadecimal digits, then only the low order four digits are used.

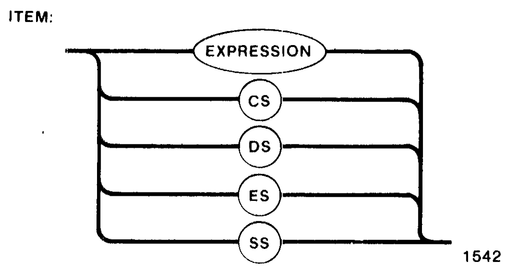


Figure 3-1. Syntax Diagram For Item

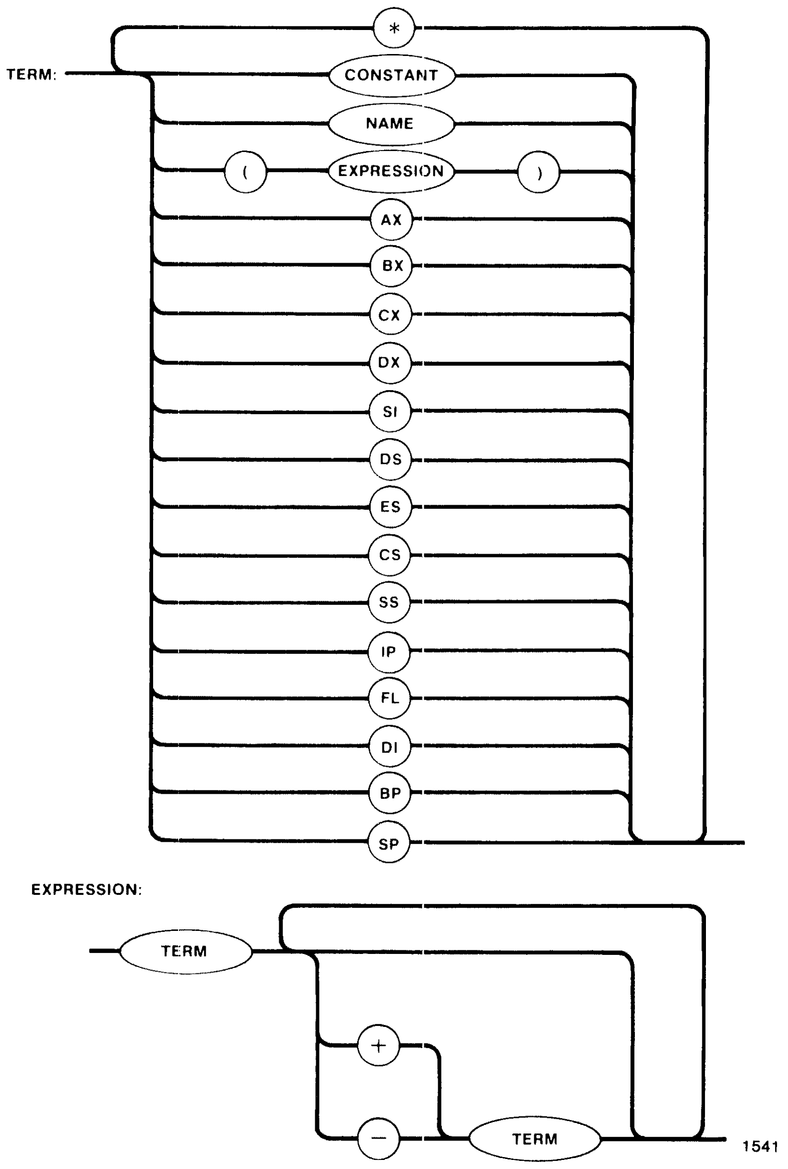


Figure 3-2. Syntax Diagrams For Term And Expression

***

CHAPTER 4
DEBUGGER COMMANDS

This chapter presents the details of the Debugger commands. It is divided into several sections, each of which describes a related group of commands. The command groupings are as follows:

Symbolic Name Commands
Breakpoint Commands
Memory Commands
Commands to Inspect iRMX 86 Objects
Commands to View Object Lists
Commands to Exit the Debugger

Each section contains a general information portion followed by detailed command descriptions.

Between this introduction and the discussions of the individual commands is a command dictionary. This dictionary, which lists the commands in alphabetical order, includes short descriptions and page numbers of the complete descriptions in this chapter. Those commands which are not associated with specific command letters are listed at the end of the dictionary.

Because the iRMX 86 operating system can run under several microprocessors, the generic term "CPU" will be used in place of the names iAPX 86, iAPX 186, iAPX 88, iAPX 188, and iAPX 286.

COMMAND DICTIONARY

SYMBOLIC NAME COMMANDS

For your convenience during debugging, the Debugger supports the use of alphanumeric variable names that stand for numerical quantities. The names and their associated values can be accessed by the Debugger from any of the following sources:

- A Debugger-maintained symbol table. The table contains name/value pairs that have been cataloged by the Debugger as numeric variables. This section describes commands for defining, changing, listing, and deleting numeric variables.

- The object directory of the current job. The current job is defined to be the job that contains the breakpoint task. (The command used to establish the breakpoint task is contained in the "Breakpoint Commands" section of this chapter.) If there is no breakpoint task, the current job is the root job.

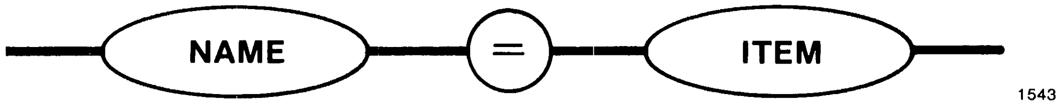
- The object directory of the root job.

When you use a symbolic name that is not the name of a breakpoint variable, the Debugger searches these sources in the order just listed.

Suppose that you want to refer to a particular task by means of the name .TASK001. If the task is cataloged in the object directory of either the root job or the current job, then the Debugger will go to the appropriate directory and fetch a token for the task whenever the name .TASK001 is used in a Debugger command. If the task is not so cataloged, you can use VJ (view job), IJ (inspect job), VT (view task), and IT (inspect task) commands to deduce a token for the task. Then you can define .TASK001 to be a numeric variable whose value is that token.

CHANGING NUMERIC VARIABLES

This command changes the value of an existing numeric variable. The syntax for this command is as follows:



PARAMETERS

NAME	Name of an existing numeric variable.
ITEM	An expression or the name of an CPU segment register. The value of ITEM is associated with the variable name NAME.

DESCRIPTION

This command removes from the Debugger symbol table the value originally associated with NAME, and replaces it with the value of ITEM.

EXAMPLES

```
.TASKA = 2F00
*
```

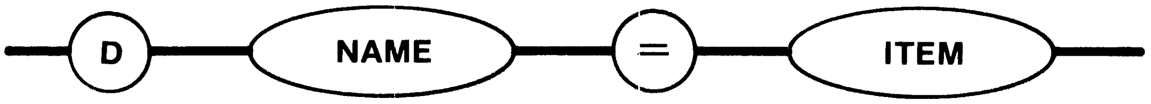
This command changes the value of .TASKA to 2F00h.

```
.TASKA = .TASKB
*
```

This command changes the value of .TASKA to that of .TASKB. In a previous example, .TASKB had a value of 2B8Ch. Therefore, this command changes the value of .TASKA to 2B8Ch.

DEFINING NUMERIC VARIABLES -- D

This command associates a variable name with a numeric value. The syntax for the D command is as follows:



1544

PARAMETERS

NAME	Name of the variable. This must be a period followed by up to 11 characters, the first of which must be alphabetic. The other characters can be alphabetic, numeric, question marks (?), or dollar signs ($).
ITEM	An expression or the name of an CPU segment register. The value of ITEM is associated with the variable name NAME.

DESCRIPTION

This command places NAME and the value of ITEM into the Debugger symbol table. You can use this command to create symbolic names for tokens, registers, or any other values. Then, you can use the symbolic names in other Debugger commands instead of entering the actual hexadecimal values.

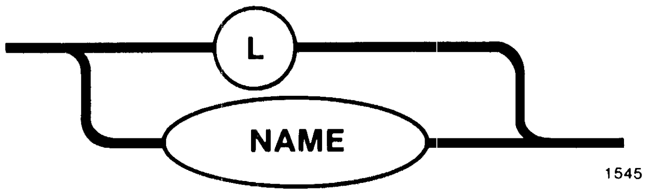
EXAMPLES

```
D .TASKA = 2DC3
*
```

This command creates a symbol called .TASKA in the Debugger's local symbol table and assigns this symbol the hexadecimal value 2DC3.

LISTING NUMERIC AND BREAKPOINT VARIABLES -- L

This command lists numeric and breakpoint variable names and their associated values. The syntax for the L command is as follows:



PARAMETER

NAME	Name of an existing numeric or breakpoint variable. If entered, the Debugger lists the name and value of the indicated name only.

DESCRIPTION

The L command lists all numeric and breakpoint variable names and their associated values. (Breakpoint variables are described in the "Breakpoint Commands" section of this chapter.) Specifying NAME instead of L causes only one pair to be listed. In either case, one pair is listed per line in the format:

```
NAME=xxxx
```

where xxxx is the associated value.

EXAMPLES

```
L
BP=2DC3:00FF
MBOX            2F34
TASKA           2DC3
TASKB           2B8C
TASKC           2D8A
TASKD           2CEF
*
```

This command lists the names and values of all the numeric and breakpoint variables in the Debugger's local symbol table. It lists one breakpoint variable (.BP) and four numeric variables (.TASKA, .TASKB, .TASKC, and .TASKD).

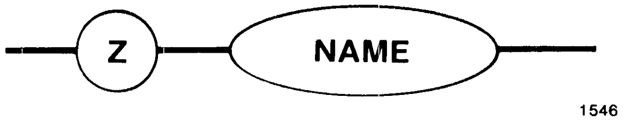
EXAMPLES (continued)

```
    .TASKA
    TASKA=2DC3
    *
```

This command lists the value associated with the variable .TASKA.

DELETING NUMERIC VARIABLES -- Z

This command deletes a numeric variable. The syntax for the Z command is as follows:



PARAMETER

NAME	Name of an existing numeric variable to be deleted.

DESCRIPTION

This command removes the NAME and associated value from the Debugger's symbol table.

EXAMPLE

```
Z .TASKA
*
```

This command deletes the numeric variable .TASKA.

BREAKPOINT COMMANDS

The Debugger provides you with the ability to set, change, view, or delete breakpoints. You set a breakpoint by defining an act which a task can perform. When a task performs the act, it incurs the breakpoint, causing its execution to cease. The Debugger supports three kinds of breakpoints:

- Execution breakpoint. A task incurs an execution breakpoint when it executes an instruction that is at a designated location in memory.
- Exchange breakpoint. A task incurs an exchange breakpoint when it performs a designated type of operation (send or receive) at a designated exchange.
- Exception breakpoint. A task incurs an exception breakpoint if its exception handler has been declared to be the Debugger and the task causes an exceptional condition of the type that invokes its exception handler.

When a task incurs a breakpoint (of any type), three things occur automatically:

- The task is placed in a pseudostate called "broken". Depending on the breakpoint options selected, the broken task and the tasks in the containing job might be suspended.
- If suspended, the broken task (and suspended tasks, if any) is (are) placed on a Debugger-maintained list called the breakpoint list. You can resume a task on the breakpoint list or you can remove it from the list.
- At the terminal, a display informs you that a breakpoint has been incurred. It also provides information about the event.

Each task on the breakpoint list is assigned a breakpoint state, which reflects the kind of breakpoint last incurred by the task. The states are as follows:

X	---	The task incurred an execution breakpoint.
E	---	The task incurred an exchange breakpoint.
Z	---	The task incurred an exception breakpoint.
N	---	The task was placed on the breakpoint list when another task in the same job incurred a breakpoint which had been set with the DB command (described later) using the J option.

You set an execution or exchange breakpoint with the DB command by defining a breakpoint variable and assigning it a breakpoint request. The request specifies to the Debugger the nature of the breakpoint, and the variable provides you with a convenient means of talking to the Debugger about the breakpoint. Using the breakpoint variable, you can cancel the breakpoint or replace it with a new one.

If you want to monitor a particular task that has not necessarily incurred a breakpoint, you can designate it to be the breakpoint task. If the task is not on the breakpoint list when you do this, the task is suspended. However, it is not placed on the breakpoint list. After designating a breakpoint task, you can examine and alter some of its registers. You can also ascertain the breakpoint state of the task. When ready, you can easily resume the task.

The Debugger displays information when a task incurs a breakpoint. The format of the display depends on the kind of breakpoint incurred.

When the task is accessing a region, the Debugger cannot process breakpoints normally. When this situation occurs, the Debugger displays the following message:

```
TASK IN REGION INCURRED BREAKPOINT:  bp-var, TASK=jjjjJ/ttttT
  FULL BREAKPOINT INFORMATION NOT AVAILABLE
  TASK NOT PLACED ON BREAKPOINT LIST
```

where:

bp-var	The name of the breakpoint variable.
jjjj	A token for the task's job.
tttt	A token for the task.

EXECUTION BREAKPOINT DISPLAY

The Debugger displays the following information when a task incurs an execution breakpoint.

```
bp-var:  E, TASK=jjjjJ/ttttq, CS=cccc, IP=iiii
```

where:

bp-var	The name of the breakpoint variable.
jjjj	A token for the task's job.

tttt	A token for the task.
q	Either T (for task) or * (indicating that the task has overflowed its stack).
cccc	The base of the code segment in which the breakpoint was set.
iiii	The offset of the breakpoint within its code segment.

EXCHANGE BREAKPOINT DISPLAY

The Debugger displays the following information when a task incurs an exchange breakpoint:

```
bp-var:  a, EXCH=jjjjJ/xxxxe, TASK=jjjjJ/ttttq, ITEM=item
```

where:

bp-var	The name of the breakpoint variable.
a	Indicates which kind of operation (S for send or R for receive) caused the breakpoint to be incurred.
jjjj	A token for the job containing the exchange whose token follows.
xxxx	A token for the exchange.
e	Indicates the type of the exchange (M for mailbox, S for semaphore, R for region).
tttt	A token for the task.
q	Either T (for task) or * (indicating that the task has overflowed its stack).
item	One of the following:

If the exchange is a mailbox, this field lists a pair of tokens, of the form:

jjjjJ/oooot,

where:

jjjj	A token for the mailbox's containing job.

oooo	A token for the object being sent or received.
t	The type of the object being sent or received (J for job, T for task, M for mailbox, S for semaphore, G for segment, R for region, X for extension, and C for composite).

If the kind of operation was receive, but no object was there to be received, item is 0000.

If the exchange is a semaphore, this field lists the number of units held by the exchange.

EXCEPTION BREAKPOINT DISPLAY

The Debugger displays the following information when a task incurs an exception breakpoint:

```
EXCEPTION:  jjjjJ/ttttT, CS=cccc, IP=iiii, TYPE=wwww, PARAM=vvvv
```

where:

jjjj	A token for the job which contains the task that caused the exception condition.
tttt	A token for the task that caused the exceptional condition.
cccc and iiii	Respectively, the contents of the iAPX 86 CS and IP registers when the exceptional condition occurred.
wwww	The numerical value of the exception code; reflects the nature of the exceptional condition. Refer to the iRMX reference manuals for the mnemonic condition codes and their numerical equivalents.
vvvv	The number (0001 for first, 0002 for second, etc.) of the parameter that caused the exceptional condition. If no parameter was at fault, vvvv is 0000.

EXCEPTION BREAKPOINT DIFFERENCES

Exception breakpoints differ from execution and exchange breakpoints in several respects:

- It is not possible to set, change, view, or delete exception breakpoints by using the commands of the Debugger. Instead, each task can set an exception breakpoint by declaring the Debugger to be its exception handler. The task can subsequently delete the breakpoint by declaring a different exception handler. However, like the other kinds of breakpoints, once a task incurs an exception breakpoint and is placed on the breakpoint list, you can cause it to resume execution with the same command (the G command) that is used to resume other tasks on the breakpoint list.

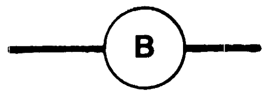
- An exception breakpoint is set for a particular task. Execution and exchange breakpoints are set for no particular task; any task can incur such a breakpoint.

- An exception breakpoint is not known to the Debugger by a breakpoint variable name.

The handling of exception breakpoints is significantly different from that of execution and exchange breakpoints. For example, exception breakpoints cannot be viewed, but the other breakpoints can be. Wherever this distinction applies, this chapter points it out.

VIEWING BREAKPOINT PARAMETERS -- B

This command displays the breakpoint parameters. The syntax for the B command is as follows:



1547

DESCRIPTION

The B command performs the following three functions:

- Displays the breakpoint list
- Displays the breakpoint task
- Displays the breakpoint variables

Breakpoint List Display

The B command first displays the breakpoint list in the following format:

```
BL=jjjjJ/ttttT(s) jjjjJ/ttttT(s) ... jjjjJ/ttttT(s)
```

where:

jjjj	A token for the job containing the task whose token follows.
tttt	A token for a task on the breakpoint list.
s	The breakpoint state of a task. Possible values are X (for execution), E (for exchange), Z (for exception), and N (for null).

Breakpoint Task Display

The second effect of the B command is to display the breakpoint task originally selected with the BT command. The format of this display is as follows:

```
    BT=jjjjJ/ttttT(s)
```

where:

jjjj	A token for the job containing the breakpoint task.
tttt	A token for the breakpoint task.
s	The breakpoint state of the breakpoint task. Possible values are X (for execute), E (for exchange), Z (for exception), and N (for null).

If there is no breakpoint task, the display is:

```
    BT=0
```

Breakpoint Variables Display

The third and final effect of the B command is to display the breakpoint variables. The format of the display depends on whether the variables are execution or exchange variables.

Execution breakpoints are displayed as:

```
    bp-var = xxxx:yyyy z ops
```

where:

bp-var	The name of the breakpoint variable.

xxxx	The base portion of the address at which the breakpoint is set.
yyyy	The offset portion of the address at which the breakpoint is set.
z	Indicates whether a task (T) or all the tasks in a job (J) are to be suspended and placed on the breakpoint list when the breakpoint is incurred.
ops	Indicates the breakpoint options. If any are present, they can be a C (for Continue task) and/or D (for Delete breakpoint).

Exchange breakpoints are displayed as:

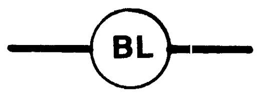
```
    bp-var = xxxx a z ops
```

where:

bp-var	The name of the breakpoint variable.
xxxx	A token for the exchange at which the breakpoint is set.
a	Indicates the kind of breakpoint activity at the exchange, either S (for Send), R (for Receive), or SR (for both).
z	Indicates whether a task (T) or all the tasks in a job (J) are to be suspended and placed on the breakpoint list when the breakpoint is incurred.
ops	Indicates the breakpoint options. If any are present, they can be C (for Continue task) and/or D (for Delete breakpoint).

VIEWING THE BREAKPOINT LIST -- BL

This command displays the breakpoint list. The syntax for the BL command is as follows:



1548

DESCRIPTION

The BL command displays the entire breakpoint list at the user terminal. This list appears as follows:

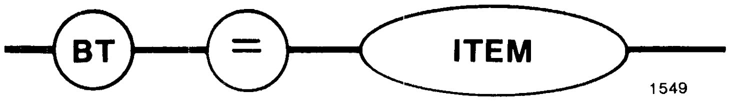
```
     BL=jjjjJ/ttttT(s) jjjjJ/ttttT(s) ... jjjjJ/ttttT(s)
```

where:

jjjj	A token for the job containing the task whose token follows.
tttt	A token for a task.
s	The breakpoint state of a task. Possible values are X (for execution), E (for exchange), Z (for exception), and N (for null).

ESTABLISHING THE BREAKPOINT TASK -- BT

This command designates a task to be the breakpoint task. The syntax for the BT command is as follows:



PARAMETER

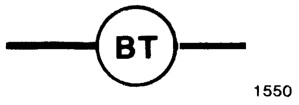
ITEM	A token for an existing task.

DESCRIPTION

The task designated by ITEM becomes the breakpoint task. The Debugger suspends the task but does not place it on the breakpoint list.

LISTING THE BREAKPOINT TASK -- BT

This command lists the job and task tokens associated with the breakpoint task. The syntax for the BT command is as follows:



DESCRIPTION

This command displays the following information about the breakpoint task:

```
    BT=jjjjJ/ttttT(s)
```

where:

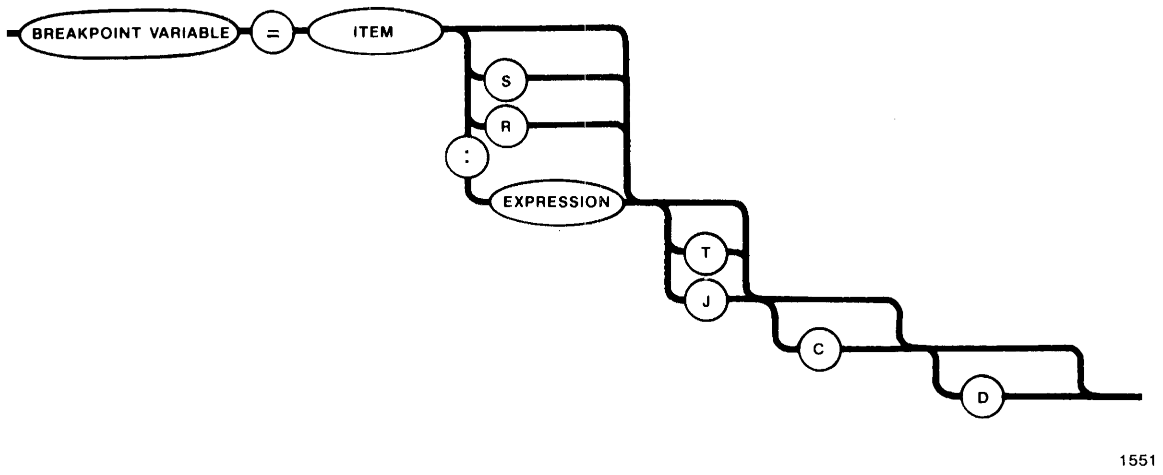
jjjj	A token for the job containing the breakpoint task.
tttt	A token for the breakpoint task.
s	The breakpoint state of the breakpoint task. Possible values are X (for execute), E (for exchange), Z (for exception), and N (for null).

If there is no breakpoint task, the Debugger displays the following:

```
    BT=
```

CHANGING A BREAKPOINT

This command changes an existing breakpoint. The syntax for this command is as follows:



PARAMETERS

BREAKPOINT VARIABLE	An existing Debugger breakpoint name. If the Debugger's symbol table does not already contain this name, an error message will appear on the terminal's display.
ITEM	If you are changing an execution breakpoint, ITEM is used in combination with EXPRESSION to specify the address of the breakpoint. ITEM must contain the base portion of the address. It must be followed by ":" and an EXPRESSION, which must contain the offset portion. If you are changing an exchange breakpoint, ITEM must contain a token for an exchange.
S and R	To be used only when changing an exchange breakpoint. S means that the exchange breakpoint is for senders only, while R means that it is for receivers only. If you want to set an exchange breakpoint for both senders and receivers, omit both S and R, as well as both ":" and EXPRESSION.

T and J	Indicate which tasks are to be put on the breakpoint list when a breakpoint is incurred. T indicates only the task that incurred the breakpoint, while J indicates all of the tasks in that task's job. If neither T nor J is present, T is assumed.
C	Continue task execution option. This option directs the Debugger not to "break" tasks that incur the breakpoint, and not to put them on the breakpoint list. When a task incurs such a breakpoint, the Debugger generates a breakpoint display, but the task continues to run.
D	Delete breakpoint option. This option directs the Debugger to delete the breakpoint after it is first incurred by a task. The Debugger generates a breakpoint display and, unless the C option is also specified, places the task that incurred the breakpoint on the breakpoint list.

DESCRIPTION

This command deletes the breakpoint that was associated with the breakpoint variable name and replaces it with a new breakpoint, as specified in the command. The breakpoint variable name can be used when deleting or changing the breakpoint.

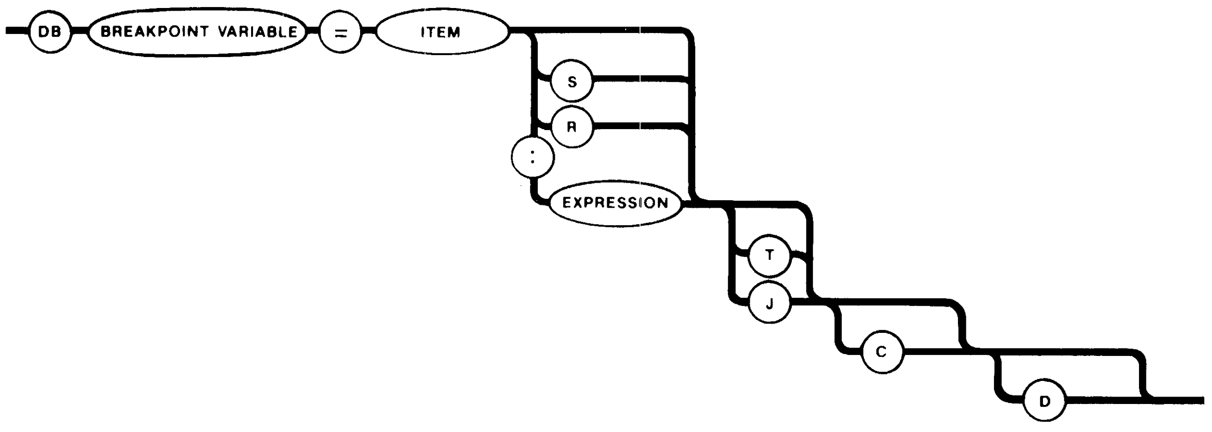
EXAMPLE

```
.BPOINT
BPOINT=2F34 S T C
*
.BPOINT = 2D2A S C
*
.BPOINT
BPOINT=2D2A S C
*
```

In this example, the user lists a breakpoint variable, changes it, and lists it again.

DEFINING A BREAKPOINT -- DB

This command defines an execution or exchange breakpoint. The syntax for the DB command is as follows:



1552

PARAMETERS

BREAKPOINT VARIABLE	A Debugger name by which to identify the breakpoint. This name must consist of a period followed by up to 11 characters, the first of which must be alphabetic. The other characters can be alphabetic, numeric, question marks (?), or dollar signs ($). If the Debugger's symbol table already contains this name, an error message will appear on the terminal's display.
ITEM	If you are setting an execution breakpoint, ITEM is used in combination with EXPRESSION to specify the address of the breakpoint. ITEM must contain the base portion of the address. It must be followed by ":" and an EXPRESSION, which must contain the offset portion. If you are setting an exchange breakpoint, ITEM must contain a token for an exchange.
S and R	To be used only when setting an exchange breakpoint. S means that the exchange breakpoint is for senders only, while R means that it is for receivers only. If you want to set an exchange breakpoint for both senders and receivers, omit both S and R, as well as both ":" and EXPRESSION.

EXPRESSION	To be used only when setting an execution breakpoint. EXPRESSION must contain the offset portion of the address of the execution breakpoint.
T and J	Indicate which tasks are to be put on the breakpoint list when a breakpoint is incurred. T indicates only the task that incurred the breakpoint, while J indicates all of the tasks in that task's job. The default is T.
C	Continue task execution option. This option directs the Debugger not to "break" tasks that incur the breakpoint, and not to put them on the breakpoint list. When a task incurs such a breakpoint, the Debugger generates a breakpoint display, but the task continues to run.
D	Delete breakpoint option. This option directs the Debugger to delete the breakpoint after it is first incurred by a task. The Debugger generates a breakpoint display and, unless the C option is also specified, places the task that incurred the breakpoint on the breakpoint list.

DESCRIPTION

The DB command sets a breakpoint of the type indicated in the remainder of the command line. The name designated as the breakpoint variable can be used when altering or deleting the breakpoint.

EXAMPLES

```
    DB .BP = 2DC3:0FF
    *
```

This command defines an execution breakpoint at address 2DC3:0FF and assigns the name .BP to this breakpoint. When a task incurs this breakpoint, only the task itself is placed on the breakpoint list.

```
    DB .BPOINT = .MBOX S C
    *
```

This command defines an exchange breakpoint at the mailbox whose token is specified by the numeric variable .MBOX. In a previous example, .MBOX had a value of 2F34; therefore the Debugger uses this value for the token.

EXAMINING A BREAKPOINT

This command displays information about a particular breakpoint. The syntax for this command is as follows:

BREAKPOINT VARIABLE
1553

PARAMETER

BREAKPOINT VARIABLE	The name of an existing breakpoint to be examined.

DESCRIPTION

The Debugger displays two kinds of output, depending on whether the specified breakpoint variable represents an execution or an exchange breakpoint. Exception breakpoints cannot be examined.

EXECUTION BREAKPOINT OUTPUT

If the designated breakpoint is an execution breakpoint, the Debugger sends the following display to the terminal:

```
     bp-var=xxxx:yyyy  z ops
```

where:

bp-var	The name of the breakpoint variable.
xxxx	Base portion of the breakpoint's address.
yyyy	Offset portion of the breakpoint's address.
z	Indicates whether a single task (T) is to be "broken" and placed on the breakpoint list or all tasks in a job (J) are to be suspended and placed on the breakpoint list, when the breakpoint is incurred.
ops	Indicates the breakpoint options. If any are present, they can be C (for Continue task) and/or D (for Delete breakpoint).

EXCHANGE BREAKPOINT OUTPUT

If the designated breakpoint is an exchange breakpoint, the Debugger sends the following display to the terminal:

```
    bp-var=xxxx a z ops
```

where:

bp-var	The name of the breakpoint variable.
xxxx	A token for the exchange at which the breakpoint is set.
a	Indicates the kind of breakpoint activity at the exchange, either S (for send), R (for receive), or SR (for both).
z	Indicates whether a single task (T) is to be "broken" and placed on the breakpoint list or all tasks in a job (J) are to be suspended and placed on the breakpoint list, when the breakpoint is incurred.
ops	Indicates the breakpoint options. If any are present, they can be C (for continue task) and/or D (for delete breakpoint).

EXAMPLES

```
    .BP
    BP=2DC3:00FF T
    *
```

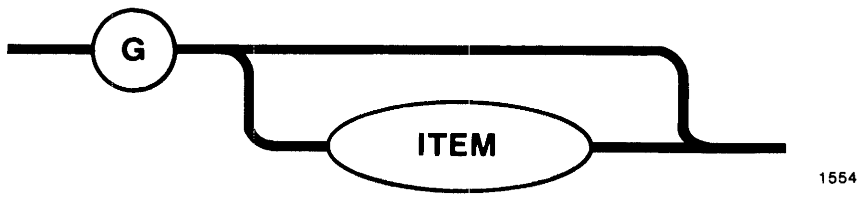
This command lists the address of the execution breakpoint associated with variable .BP. It also indicates that only the task is to be "broken" if a breakpoint is encountered.

```
    .BPOINT
    BPOINT=2F34 S T C
    *
```

This command lists the address of the exchange breakpoint associated with variable .BPOINT. The S, T, and C indicate that only tasks which send messages to the exchange will incur the breakpoint, only the task that incurs the breakpoint will be "broken", and the task will continue processing after incurring the breakpoint.

RESUMING TASK EXECUTION -- G

This command resumes execution of a task on the breakpoint list or the breakpoint task. The syntax for the G command is as follows:



PARAMETER

ITEM	A token for a task on the breakpoint list or the breakpoint task. If the given token is not for a task on the breakpoint list or the breakpoint task, an error message will be displayed. If this parameter is omitted, the breakpoint task is assumed.

DESCRIPTION

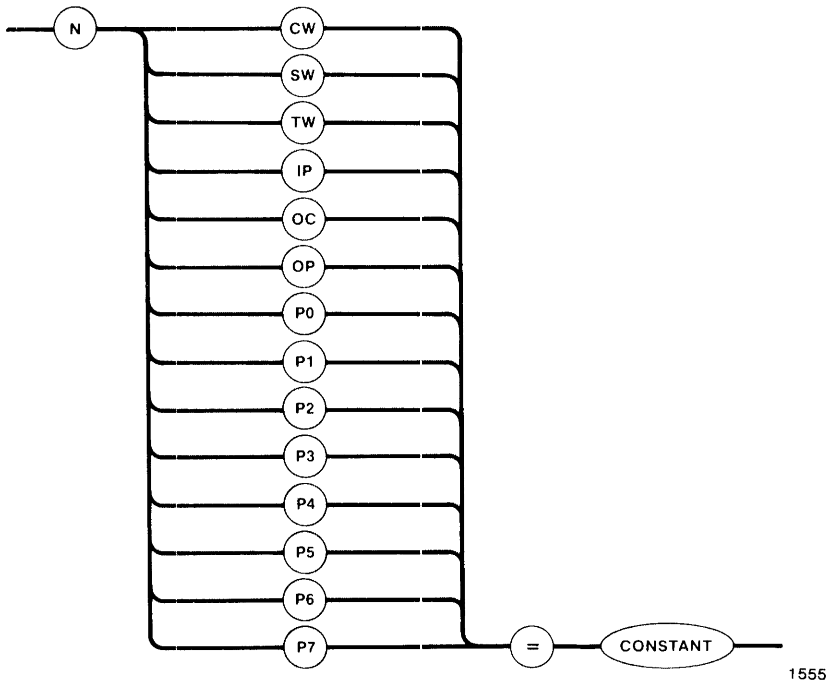
The G command applies to the breakpoint task if ITEM is not present. Otherwise, it applies to the task on the breakpoint list whose token is represented by ITEM.

The G command resumes execution of the designated task. If the task is in the broken state, it is made ready. If it is in the suspended state, its suspension depth is decreased by one.

If the G command is invoked without ITEM when there is no breakpoint task, an error message is displayed at the terminal.

ALTERING THE BREAKPOINT TASK'S NPX REGISTERS -- N

This command modifies the breakpoint task's Numeric Data Processor (NPX) register values. This command applies only to tasks that were specified at creation as having the ability to use the NPX. The syntax for this command is as follows:



PARAMETERS

CW, SW, TW, IP, OC, OP, P0 through P7 — Names of the breakpoint task's NPX registers, as follows:

Name	Description
CW	Control Word
SW	Status Word
TW	Tag Word
IP	Instruction Pointer
OC	Operation Code
OP	Operand Pointer
P0-P7	Stack elements

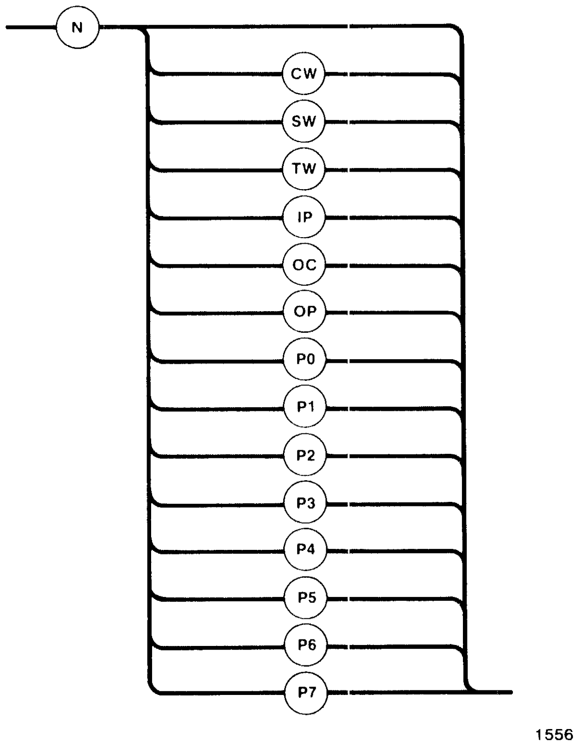
CONSTANT	A hexadecimal number which is used for the new register value. CONSTANT can specify an 80-bit value for registers P0 through P7, a 20-bit value for registers IP and OP, and a 16-bit value for the remaining registers. If this value is too large to fit in the specified register, the Debugger displays a SYNTAX ERROR message.

DESCRIPTION

This command requests that the breakpoint task's NPX register, as specified in the command request, be updated with the value of CONSTANT. This command applies only to tasks which were specified at creation as using the NPX.

VIEWING THE BREAKPOINT TASK'S NPX REGISTERS -- N

This command displays the breakpoint task's Numeric Data Processor (NPX) register values. This command applies only to tasks that were specified at creation as having the ability to use the NPX. The syntax for this command is as follows:



PARAMETERS

CW, SW, TW, IP, OC, OP, P0 through P7 — Names of the breakpoint task's NPX registers, as follows:

Name	Description
CW	Control Word
SW	Status Word
TW	Tag Word
IP	Instruction Pointer
OC	Operation Code
OP	Operand Pointer
P0-P7	Stack elements

If no name is specified, the Debugger displays values for all registers.

DESCRIPTION

This command lists NPX register values for the breakpoint task. It applies only to tasks which were specified at creation as using the NPX. If the command is simply "N", then all of the breakpoint task's NPX registers are displayed, in the following format:

```
NCW = xxxx       NSW = xxxx       NTW = xxxx
NIP = xxxxx      NOC = xxx        NOP = xxxxx
NP0 = xxxxxxxxxxxxxxxxxxxx
NP1 = xxxxxxxxxxxxxxxxxxxx
NP2 = xxxxxxxxxxxxxxxxxxxx
NP3 = xxxxxxxxxxxxxxxxxxxx
NP4 = xxxxxxxxxxxxxxxxxxxx
NP5 = xxxxxxxxxxxxxxxxxxxx
NP6 = xxxxxxxxxxxxxxxxxxxx
NP7 = xxxxxxxxxxxxxxxxxxxx
NES = xxxx
```

The size of the field indicates the number of hexadecimal digits that the Debugger displays.

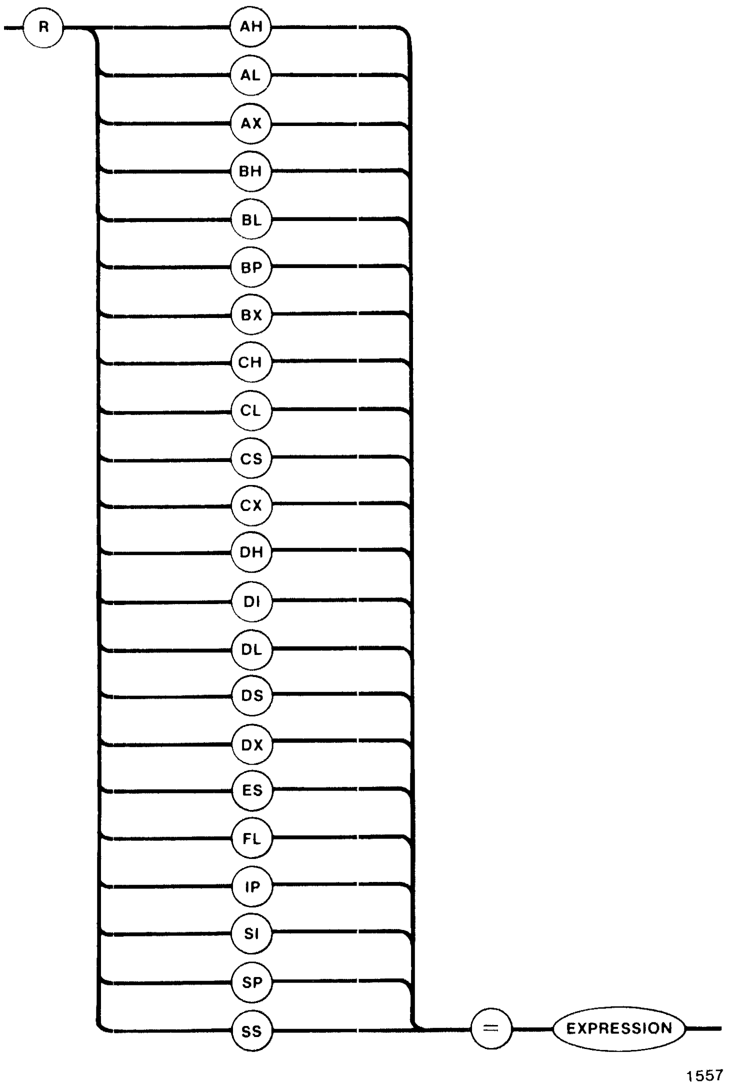
Registers P0 through P7 are 80-bit registers that the Debugger displays in temporary real format.

The NES field contains the value of the NPX Status Word if an NPX exception caused the breakpoint task to be broken. The value for this field, under all other circumstances, is NONE.

If the breakpoint task does not use the NPX, the Debugger returns an error message in response to this command.

ALTERING THE BREAKPOINT TASK'S REGISTERS -- R

This command alters one of the breakpoint task's CPU register values. The syntax for this command is as follows:



PARAMETERS

AH, AL, AX, BH, BL, BP, BX, CH, CL, CS, CX, DH, DI, DL, DS, DX, ES, FL, IP, SI,SP, SS	Names of the breakpoint task's CPU registers.

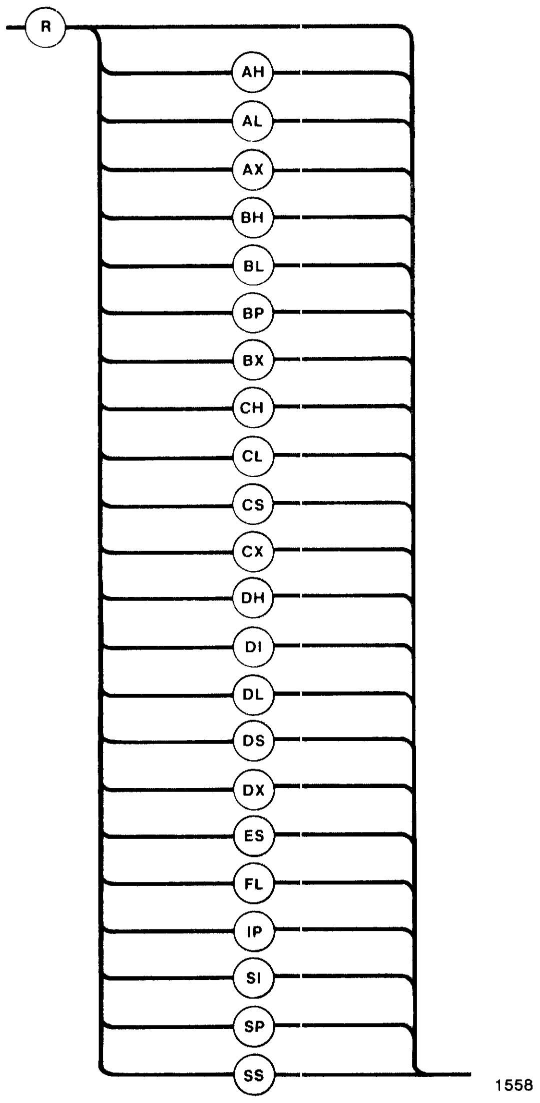
EXPRESSION	A Debugger expression whose value is used for the new register value. If this value is too large to fit in the designated register, the Debugger fills the register with the low-order bytes of the value.

DESCRIPTION

This command requests that the breakpoint task's register, as specified in the command request, be updated with the value of the EXPRESSION. However, if the breakpoint task is in the null breakpoint state, its register values cannot be altered by the R command.

VIEWING THE BREAKPOINT TASK'S REGISTERS -- R

This command lists one or all of the breakpoint task's CPU registers. The syntax for the R command is as follows:



PARAMETERS

AH, AL, AX, BH, BL, BP, BX, CH, CL, CS, CX, DH, DI, DL, DS, DX, ES, FL, IP, SI, SP, SS	Names of the breakpoint task's CPU registers. If no name is specified, the Debugger displays values for all registers.

DESCRIPTION

This command lists CPU register values for the breakpoint task. If the command is simply "R", then all of the breakpoint task's registers are displayed, in the following format:

```
RAX=xxxx    RSI=xxxx    RCS=xxxx    RIP=xxxx
RBX=xxxx    RDI=xxxx    RDS=xxxx    RFL=xxxx
RCX=xxxx    RBP=xxxx    RSS=xxxx
RDX=xxxx    RSP=xxxx    RES=xxxx
```

If the command has the form Ryy, where yy is the register name, then the contents of the specified register are displayed, either as:

```
Ryy=xxxx
```

or as:

```
Ryy=xx
```

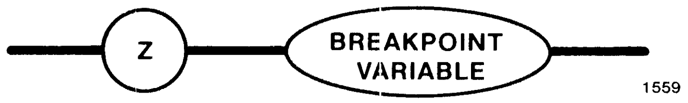
depending on whether yy is a byte-size register (like AH) or a word-size register (like AX).

If the breakpoint task is in the null breakpoint state, only its BP, SP, CS, DS, SS, IP, and FL register contents are displayed. The remaining register displays consist of question marks.

In certain circumstances the breakpoint task, when suspended, is in a state which prevents the Debugger from obtaining its register contents. If this is the case, the Debugger displays question marks for all registers.

DELETING A BREAKPOINT -- Z

This command deletes a breakpoint. The syntax for the Z command is as follows:



PARAMETER

BREAKPOINT VARIABLE	Name of an existing Debugger breakpoint to be deleted.

DESCRIPTION

The Z command deletes the specified breakpoint and removes the breakpoint variable name from the Debugger's symbol table.

EXAMPLE

```
Z .BP
*
```

This command deletes the breakpoint associated with the variable .BP and removes .BP from the Debugger's symbol table.

MEMORY COMMANDS

The commands in this section enable you to inspect or modify the contents of absolute memory locations. Figure 4-1 illustrates the syntax for all commands in this section.

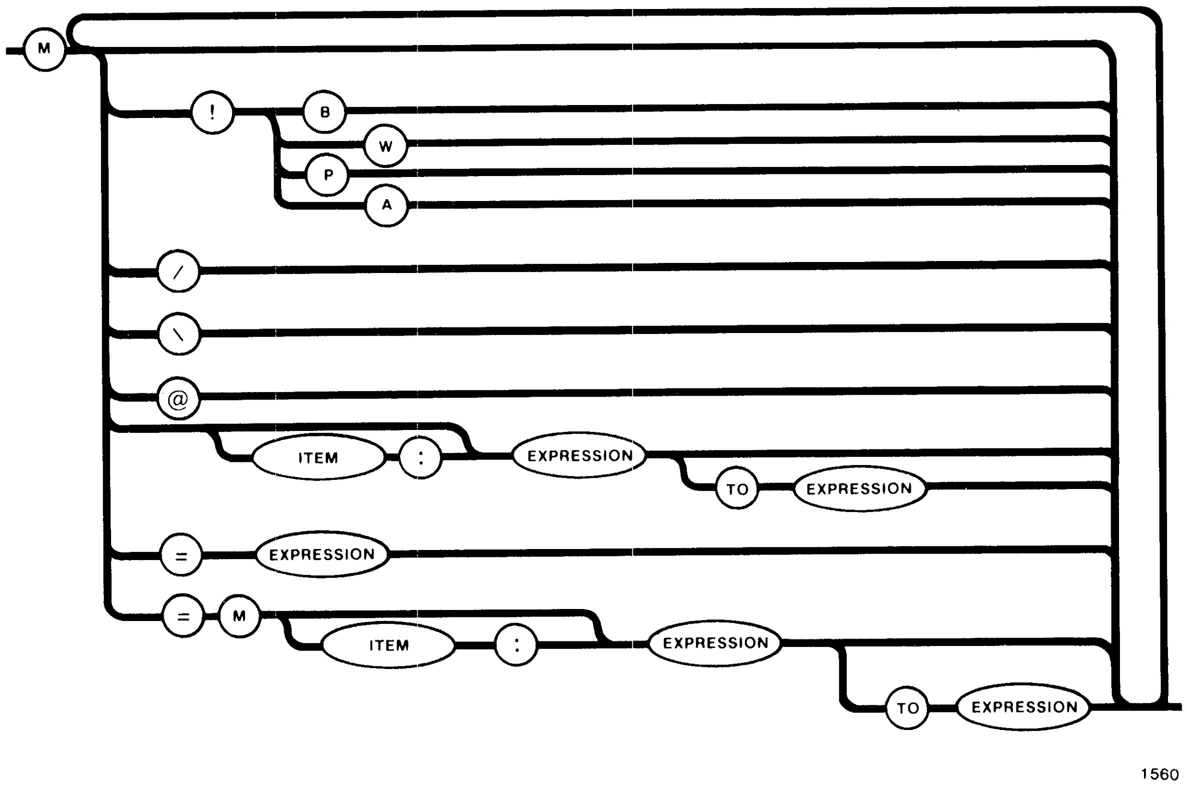


Figure 4-1. Syntax Diagram for Memory Commands

As Figure 4-1 illustrates, all memory commands begin with "M". There are a variety of parameters that can be specified with "M"; these parameters are grouped into the following basic options:

- Setting current display mode. This option begins with "!".
- Changing memory locations. This option includes the "=".
- Displaying memory locations. This option consists of the remaining parameters.

This section breaks up the description of the "M" command into these three groups and discusses the groups as separate commands. However, you can combine any number of "M" command options in a single command, as the syntax diagram in Figure 4-1 illustrates.

In the descriptions of these commands, frequent mention is made of the current display mode, the current segment base, the current offset, the current address, and the display of memory locations. This terminology is defined as follows:

- The current display mode determines the manner in which memory values are interpreted for display purposes. The possible modes are designated by the letters B, W, P, and A, and they stand, respectively, for byte, word, pointer, and ASCII. The effects of these modes are best explained in the context of an example. Suppose that memory locations 042B through 042E contain, respectively, the values 25, F3, 67, and 4C. If you ask for the display of the memory at location 042B, then the effects, which depend on the current display mode, are as follows:

Current Display Mode	Display
B	25
W	F325
P	4C67:F325
A	%

  Observe that words and pointers are displayed from high-order (high address) to low-order (low address).

  If a location contains a value which does not represent a printable ASCII character, and the current display mode is A, then the Debugger prints a period. The initial current display mode is B.

- The value of the current segment base is always the value of the most recently used CPU segment base. The initial value of the current segment base is 0.

- The current offset is a value the Debugger maintains and uses when reference is made to a memory location without explicitly citing an offset value. Except when the current offset has been modified by certain options of the M command, the current offset is always the value of the most recently used offset. The initial value of the current offset is 0.

- The current address is the iAPX 86 memory address computed from the combination of the current segment base and the current offset.

- When memory locations are displayed, the format is as follows:

```
    xxxx:yyyy=value
```

  where xxxx and yyyy are the current segment base and current offset, respectively, and value is a byte, word, pointer, or ASCII character, depending on the current display mode. If several contiguous memory locations are being requested in a single request, each line of display is as follows:

```
    xxxx:yyyy=value value value ... value
```

  where xxxx, yyyy, and value are as previously described, and xxxx:yyyy represent the address of the first value on that line.

  The first such line begins with the first address in the request and continues to the end of that (16-byte) paragraph. If additional lines are required to satisfy the request, each of them begins at an offset which is a multiple of 16 (10 hexadecimal).

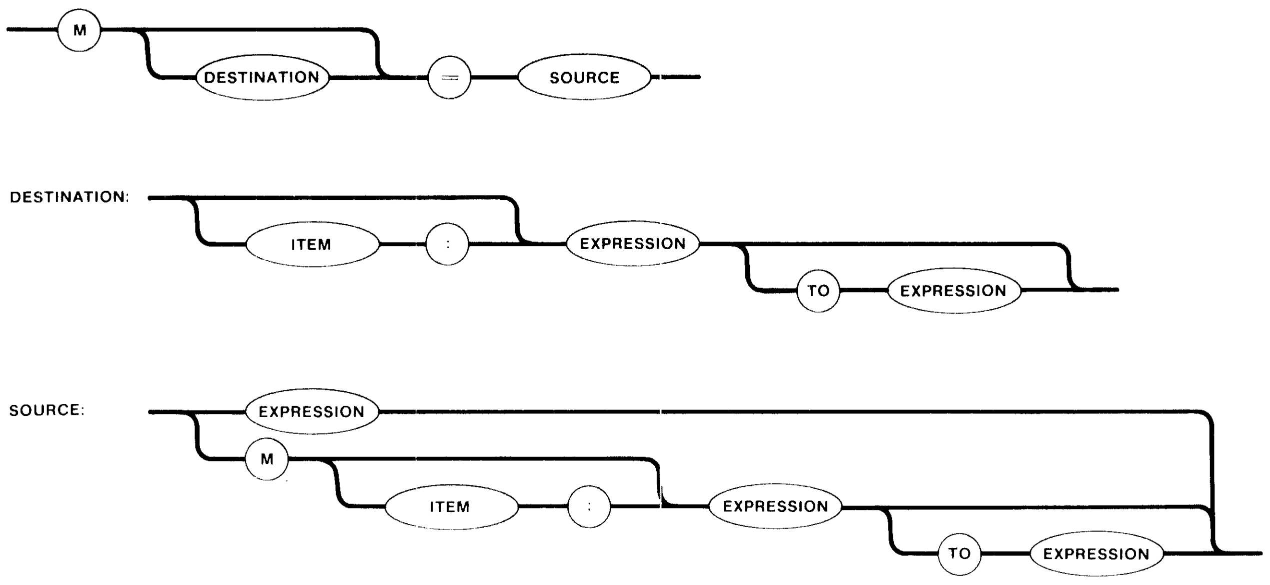
CHANGING MEMORY -- M

This command changes the contents of designated RAM locations.

CAUTION

Because the Debugger is generally used during system development, while your tasks, the Nucleus, the Debugger, and possibly other iRMX 86 components are in RAM, you should use these M command options with extreme care.

The syntax for this command is as follows:



1561

PARAMETERS

As shown in the syntax diagram, the parameters for this command are divided into DESTINATION and SOURCE parameters which are separated with an equal sign.

Destination Parameters

These parameters define the memory location or locations that are going to be changed. All parameters change the current offset, and some of them change the current base. The valid parameter combinations are as follows:

EXPRESSION	This form of the DESTINATION option implies that the address to be changed has the current base as its base value and the value of EXPRESSION as its offset.
ITEM:EX-PRESSION	This form of the DESTINATION option implies that the address to be changed has the value of ITEM as its base value and the value of EXPRESSION as its offset.
EXPRESSION TO EXPRESSION	This form of the DESTINATION option implies that a series of consecutive locations will be changed. The EXPRESSIONs determine the beginning and ending offsets, respectively. The current base is used as a base value. After memory has been changed, the current offset is set to the value of the second EXPRESSION.
ITEM:EX-PRESSION TO EXPRESSION	This form of the DESTINATION option is the same as the previous one, except that ITEM is used as the base value of the locations.

If no DESTINATION option is specified, the location specified by the current segment base and current offset is changed. However, if the previous command was a "Display Memory" command of the form:

```
M EXPRESSION TO EXPRESSION
```

the entire range of locations specified in that command is changed.

Source Parameters

These parameters define the information that will be placed into the DESTINATION memory. The valid parameter combinations are as follows:

EXPRESSION — This form of the SOURCE option can be used only if the current display mode is byte or word. It implies that the value represented by EXPRESSION will be copied into the byte or word at the current address. However, if the DESTINATION option (supplied or default) specified a range of locations, this option instead copies the value of EXPRESSION into each byte or word in DESTINATION.

Examples:

(1) When the DESTINATION option did not specify a range of values:

```
M = 4C
0400:0008 09
0400:0008 4C
*
```

This example changes the contents of the current location (0400:0008) from 09 to 4C. Notice that the Debugger displays both the old and the new contents of memory.

(2) When the DESTINATION option specified a range of values:

```
M 1 TO 4
0400:0001 06 07 08 09
*
M = 4C
0400:0001 06 07 08 09
0400:0001 4C 4C 4C 4C
*
```

In this example, because the previous command was an examination of a range of memory, the command to change memory changes the entire range of memory.

M EXPRESSION — This form of the SOURCE option uses the current segment base and the offset indicated by the value of EXPRESSION to compute an address. It copies the value at that computed address into the location specified by the current address.

However, if the DESTINATION option (supplied or default) specified a range of locations, the value at the computed address is instead copied to each of the locations in the destination field.

Examples:

(1) When the DESTINATION option did not specify a range of values:

```
     M 9
     0400:0009 11
     *
     M = M 6
     0400:0009 11
     0400:0009 4C
     *
```

This example replaces the value in location 4000:0009 (11) with the value in location 4000:0006 (4C).

(2) When the DESTINATION option specified a range of values:

```
     M 100
     0400:0100 FF
     *
     M 100 TO 103 = M 6
     0400:0100 FF A0 16
     0400:0100 4C 4C 4C
     *
```

In this example, the command to change memory included a DESTINATION option that specified a range of values. Thus the contents of location 0400:0006 (4C) are copied into each of the DESTINATION locations.

M ITEM:EXPRESSION — This form of the SOURCE option uses ITEM and EXPRESSION as base and offset, respectively, to compute an address. It copies the value at that computed address into the location specified by the current address. However, if the DESTINATION option (supplied or default) specified a range of locations, the value at the computed address is instead copied to each of the locations in the destination field.

Examples:

(1) When the DESTINATION option did not specify a range of values:

```
M 9
0400:0009 4C
*
M = M 300:2643
0400:0009 4C
0400:0009 21
*
```

This example takes the value in location 0300:2643 (21) and copies it into the current location (0400:0009).

(2) When the DESTINATION option specified a range of values:

```
M 100 TO 103 = M 300:2643
0400:0100 4C 4C 4C 22
0400:0100 21 21 21 21
*
```

This example copies the contents of location 0300:2643 (21) into each of the locations specified in the DESTINATION option.

M EXPRESSION TO EXPRESSION	This form of the SOURCE option uses the current segment base and, in order, the offsets indicated by the EXPRESSIONs, to compute a beginning address and an ending address. It copies the sequence of values bounded by the computed addresses to the sequence of locations that begin at the current address. However, if the DESTINATION option (supplied or default) specified a range of locations, the sequence of values bounded by the computed addresses is copied to the destination field, with the source values being truncated or repeated as required.

Examples:

(1) When the DESTINATION option did not specify a range of values:

```
M 400:104
0400:0104 E1
*
M = M A TO C
0400:0104 E1 F2 0A
0400:0104 0B 0C 0D
*
```

In this example, the contents of the range of locations specified in the SOURCE option (0400:000A - 0400:000C) are copied into the range of locations that begin with the current address (0400:0104).

(2) When the destination option specified a range of values:

```
M 1 TO 4 = M A TO C
0400:0001 4C 4C 4C 4C
0400:0001 0B 0C 0D 0B  (first value
*                        repeated)
```

This example copies the contents of three locations (0400:000A - 0400:000C) into four locations (0400:0001 - 0400:0004). Notice that the values start repeating; 0400:0001 contains the same value as 0400:0004 (0B).

M ITEM:EXPRESSION TO EXPRESSION — This form of the SOURCE option uses ITEM as a base and the EXPRESSIONs as offsets to compute a beginning and an ending address. The sequence of values bounded by the computed addresses is copied to the sequence of locations beginning at the current address. However, if the DESTINATION option (supplied or default) specified a range of values, the sequence of values bounded by the computed addresses is copied to the destination field, with the source values being truncated or repeated as required.

Examples:

(1) When the DESTINATION option did not specify a range of values:

```
D .VALUE = 2643
*
M 1
0400:0001 0B
*
M = M 300:.VALUE TO .VALUE + 4
0400:0001 0B 0C 0D 0B 4C
0400:0001 21 47 E2 C8 31
*
```

In this example, the contents of the range of locations specified in the SOURCE option (0300:2643 - 0300:2647) are copied into the range of locations that begin with the current address (0400:0001).

(2) When the DESTINATION option specified a range of values:

```
     M 101 TO 104
     0400:0101 21 21 21 0B
     *
     M = M 300:2643 TO 2647
     0400:0101 21 21 21 0B
     0400:0101 21 47 E2 C8  (last value
     *                       truncated)
```

This example copies the contents of five locations (0300:2643 - 0300:2647) into four locations (0400:0101 - 0400:0104). Notice that the value of the fifth location (0300:2647) is not copied.

DESCRIPTION

This command changes the contents of designated RAM locations. The DESTINATION options affect the current segment base and offset values. The SOURCE options do not affect these values.

When executing this command, the Debugger displays the contents of the designated locations, then updates the contents, and finally displays the new contents. Thus, if you inadvertently destroy some important data, the information you need to restore it is available.

This command copies data in the byte mode. The current display mode is not affected by these copying options.

NOTE

When using the M command, be aware of the following hazards:

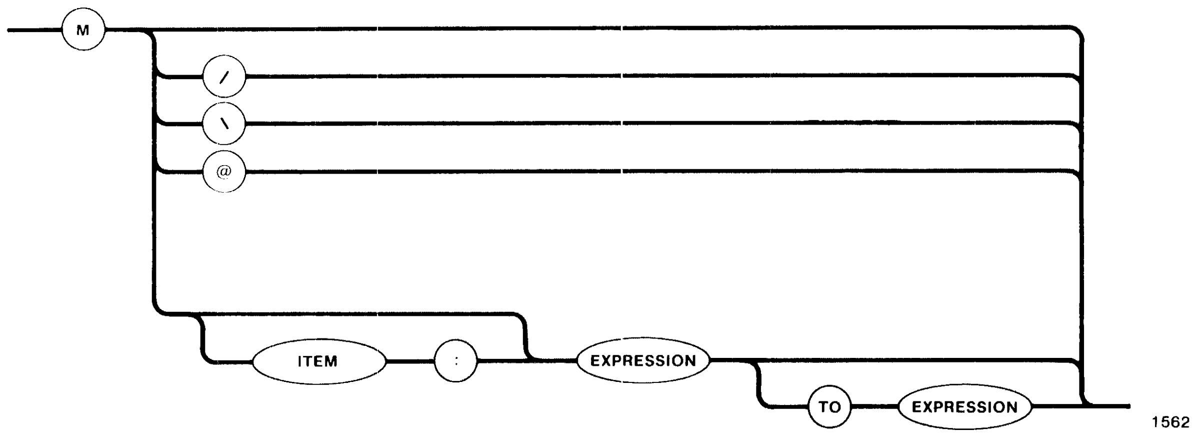
- It is possible for you to modify memory within iRMX 86 components, such as the Nucleus and Debugger. Doing so can jeopardize the integrity of your application system, and should therefore be avoided.

- It is possible to request that non-RAM memory locations be modified. If you attempt to read or write to a non-RAM location, nothing happens to memory and the displays indicate that the specified locations contain zeros.

- A memory request might cross segment boundaries. In processing such a request, the Debugger ignores such boundaries, so don't assume that a boundary will terminate a request.

EXAMINING MEMORY -- M

This command displays memory locations without changing their contents. The syntax for this command is as follows:



PARAMETERS

To avoid confusion, this section lists examples of complete commands in explaining the parameters.

M/ This option increments the current offset according to the current display mode: by one for byte or ASCII, by two for word, or by four for pointer. Then it displays the contents of the new current address.

```
Example:  M/
          0400:0009 0A
          *
```

This example increments the current offset and displays the address and contents of the location.

M This option is just like M/, except that the current offset is decremented.

```
Example:  M
          0400:0008 08
          *
```

This example decrements the current offset and displays the address and contents of the location.

M

When used by itself, M is an abbreviated way of specifying M/ or M , whichever was used most recently. If neither has been used in the current Debugging session, M is interpreted as an M/ request.

```
Example:  M
          0400:0007 08
          *
          M
          0400:0006 07
          *
```

Since M was used most recently, these commands decrement the current offset before displaying the address and contents of memory.

M@

This option sets the current offset equal to the value of the word beginning at the current address. Then the value at the adjusted current address is displayed.

```
Example:  M!B
          *
          M@
          0400:0807 46
          *
```

Even though byte mode was selected, this example sets the current offset equal to contents of the word at offset 07. From the previous example you can see that this word is indeed 0807.

M EXPRESSION

This option sets the current offset equal to the value of the EXPRESSION and displays the value at the new current address.

```
Example:  M 3
          0400:0003 04
          *
```

This example sets the current offset to 3 and displays the contents of that location.

M ITEM:EX-
PRESSION

This option is just like M EXPRESSION, except that ITEM is used as the base in the address calculation, and after the operation ITEM is the new current segment base.

```
Example:  M 300:2644
          0300:2644 47
          *
```

This example sets the current base to 300 and the current offset to 2644. It also displays the contents of that location.

M EXPRESSION TO EXPRESSION

This option displays the values of a series of consecutive locations. The EXPRESSIONs determine the beginning and ending offsets, respectively; the second EXPRESSION must be greater than the first. The current segment base is used as a base. After displaying the locations, the Debugger sets the current offset to the value of the second expression. If the specified range of locations is incompatible with the current display mode --- for example, an odd number of locations is not compatible with the word or pointer modes --- then all words or pointers that lie partially or totally inside the range are displayed.

```
Examples: (1) M 4 TO 6
              0300:0004 15 26 37
               *
          (2) M!W
               *
              M 4 TO 6
              0300:0004 2615 4837
               *
```

These examples display a consecutive series of memory locations in both byte and word mode. Notice that the base set in the last example (300) is still used.

M ITEM:EXPRESSION TO EXPRESSION

This option is just like M EXPRESSION TO EXPRESSION, except that ITEM is used as a base in the address calculation, and after the operation ITEM is the new segment base.

```
Example:  M!B
          *
          D .MEM = 100
          *
          M 400:.MEM TO .MEM +4
          0400:0100 FF A0 16 22 E1
          *
```

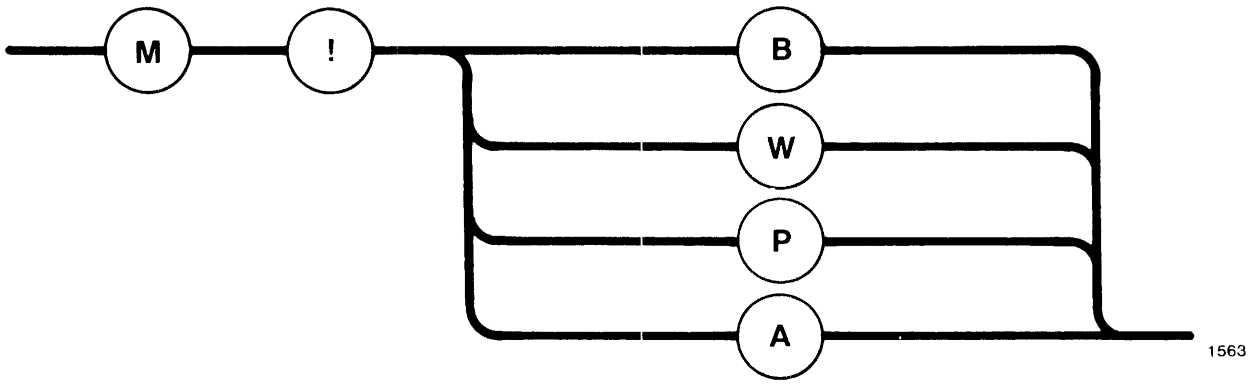
After setting the output mode to byte and defining a numeric variable .MEM, this example sets the base to 400 and displays five consecutive memory locations beginning with offset 100 (.MEM). Upon completion of this example, the current offset is 400 and the current base is 104.

DESCRIPTION

This command displays the contents of memory without disturbing those contents. Be aware, however that all of the options change the current offset, and some of them change the current segment base. None changes the current display mode.

SETTING THE CURRENT DISPLAY MODE -- M

This command specifies the way in which the Debugger will display output. The syntax for the M command is as follows:



PARAMETERS

!	Indicates that the display mode is being changed.
B, W, P, A	Specifies the mode of display. B indicates byte mode, W indicates word mode, P indicates pointer mode, and A indicates ASCII mode.

DESCRIPTION

This command sets the display mode for further Debugger output. When the Debugger next displays memory, it will display the memory according to the mode specified with this command.

EXAMPLES

```
    M!B
    *
```

This command instructs the Debugger to display all further output in byte mode.

```
    M!W
    *
```

This command instructs the Debugger to display all further output in word mode.

COMMANDS TO INSPECT SYSTEM OBJECTS

The commands in this section allow you to examine iRMX 86 objects in detail. They give specific information about the Nucleus object types. Figure 4-2 illustrates the general syntax for all the commands in this section.

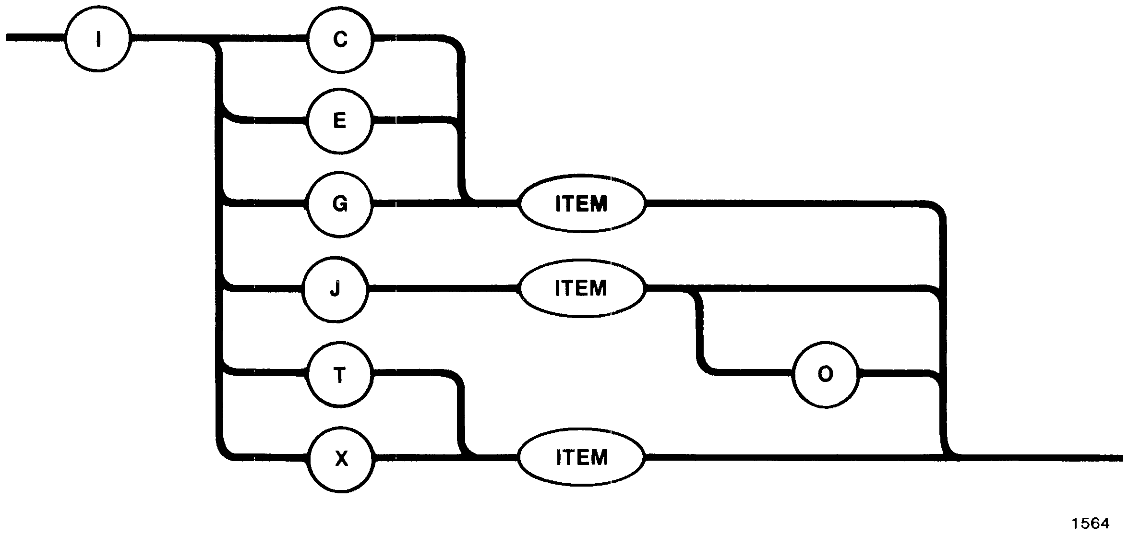


Figure 4-2. Syntax Diagram For Inspecting System Objects

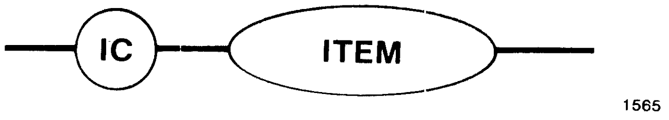
The second letter of the command indicates the type of object to inspect, as follows:

J	Job
T	Task
E	Exchange
G	Segment
C	Composite
X	Extension

The remainder of this section describes the commands in detail.

INSPECTING A COMPOSITE -- IC

This command displays the principal attributes of the specified composite. The syntax for the IC command is as follows:



PARAMETER

ITEM	Token for the composite object to be inspected.

DESCRIPTION

The IC command displays the principal attributes of the composite object whose token is represented by ITEM. Figure 4-10 depicts the form of the display produced by IC.

```
                  ----- iRMX86 COMPOSITE REPORT -----
COMPOSITE TOKEN         bbbb           CONTAINING JOB        gggg
EXTENSION TOKEN         cccc           # TOKEN SLOTS         hhhh
TOKEN(S)     ffffJ/dddde  ffffJ/dddde  ffffJ/dddde  ffffJ/dddde

NAME(S)   aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
            aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
```

Figure 4-3. An iRMX™ 86 Composite Report

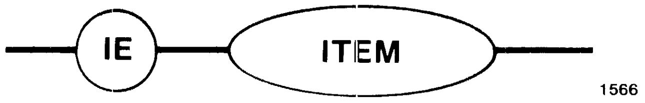
The following describes the fields in Figure 4-3:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the composite is cataloged in the object directory of either the job containing the composite or the root job. If the composite is not cataloged in either directory, "NONE FOUND" is displayed here.
bbbb	Hexadecimal token for the composite.

Field	Meaning
cccc	Hexadecimal token for the extension that represents license to create this type of composite.
dddd	Hexadecimal token for one of the components of the composite object.
e	Single letter that indicates the type of object dddd. This field can have any of the following values: C composite G segment J job M mailbox R region S semaphore T task X extension * a task whose stack has overflowed or whose code was loaded by the iRMX 86 Application Loader
ffff	Hexadecimal token for the job that contains object dddd.
gggg	Hexadecimal token for the job that contains composite object bbbb.
hhhh	Hexadecimal value specifying the maximum allowable number of component objects that the composite object can comprise.

INSPECTING AN EXCHANGE -- IE

This command displays the principal attributes of a mailbox, semaphore, or region whose token is specified. The syntax of the IE command is as follows:



PARAMETER

ITEM	Token for the exchange to be inspected.

DESCRIPTION

The IE command displays the principal attributes of the mailbox, semaphore, or region whose token is represented by ITEM. It produces three kinds of output, one for each kind of exchange.

Mailbox Display

Figure 4-4 depicts the form of display produced by IE for a mailbox.

```
                ----- iRMX86 MAILBOX REPORT -----
    MAILBOX TOKEN          bbbb         CONTAINING JOB          hhhh
    # TASKS WAITING        cccc         # OBJECTS WAITING       iiii
    FIRST WAITING   ddddf/eeeef         QUEUE DISCIPLINE     jjjjjjj
    CACHE SIZE             gggg

    NAME(S)  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
               aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
```

Figure 4-4. An iRMX™ 86 Mailbox Report

The following describes the fields in Figure 4-4:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the mailbox is cataloged in the object directory of either the mailbox's containing job or the root job. If the mailbox is not cataloged in either directory, "NONE FOUND" is displayed here.
bbbb	Hexadecimal token for the mailbox.
cccc	Number, in hexadecimal, of tasks in the mailbox's task queue.
dddd	Token for the containing job of either the first task waiting in the task queue or the first object waiting in the object queue. Because at least one of these queues is empty, dddd is not ambiguous. If both queues are empty, dddd is absent.
eeee	Token for either the first task waiting in the task queue or the first object waiting in the object queue. Because at least one of these queues is empty, eeee is not ambiguous. If both queues are empty, eeee is 0000.
f	Single letter that indicates the type of the first task waiting in the task queue or the first object waiting in the object queue. Because at least one of these queues is empty, f is not ambiguous. If both queues are empty, f is blank. Otherwise, f has one of the following values: C composite; G segment; J job; M mailbox; R region; S semaphore; T task; X extension
gggg	Number, in hexadecimal, of objects that the mailbox's high performance object queue is capable of holding.
hhhh	Hexadecimal token for the job containing the mailbox.

Field	Meaning
iiii	Number, in hexadecimal, of objects in the mailbox's object queue.
jjjjjjj	Description of the manner in which waiting tasks are queued in the mailbox's task queue. The possible values are FIFO and PRIORITY.

Semaphore Display

Figure 4-5 depicts the form of the display produced by IE for a semaphore.

```
                ----- iRMX86 SEMAPHORE REPORT -----
SEMAPHORE TOKEN        bbbb          CONTAINING JOB        gggg
# TASKS WAITING        cccc          QUEUE DISCIPLINE  hhhhhhhh
CURRENT VALUE          dddd          MAXIMUM VALUE         iiii
FIRST WAITING     eeeeJ/ffffT

NAME(S)  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
         aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
```

Figure 4-5. An iRMX™ 86 Semaphore Report

The following describes the fields in Figure 4-5:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the semaphore is cataloged in the object directory of either the semaphore's containing job or the root job. If the semaphore is not cataloged in either directory, "NONE FOUND" is displayed here.
bbbb	Hexadecimal token for the semaphore.
cccc	Number, in hexadecimal, of tasks waiting in the queue.
dddd	Number, in hexadecimal, of units currently in the custody of the semaphore.
eeee	Hexadecimal token for the containing job of the first waiting task. It is absent if no tasks are waiting.

Field	Meaning
ffff	Hexadecimal token for the first waiting task. It is 0000 if no tasks are waiting.
gggg	Hexadecimal token for the semaphore's containing job.
hhhhhhhh	Description of the manner in which waiting tasks are queued in the semaphore's task queue. The possible values are FIFO and PRIORITY.
iiii	Maximum allowable number, in hexadecimal, of units that the semaphore may have in its custody.

Region Display

Figure 4-6 depicts the form of display produced by IE for a region.

```
                    ----- iRMX86 REGION REPORT -----
   REGION TOKEN            bbbb       CONTAINING JOB          eeee
   # TASKS WAITING         cccc       QUEUE DISCIPLINE    ffffffff
   TASK IN REGION          dddd       FIRST WAITING           gggg

   NAME(S)  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
            aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
```

Figure 4-6. An iRMX™ 86 Region Report

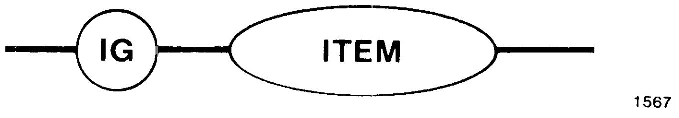
The following describes the fields in Figure 4-7:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the region is cataloged in the object directory of either the job containing the region or the root job. If the region is not cataloged in either directory, "NONE FOUND" is displayed here.
bbbb	Hexadecimal token for the region.
cccc	Number, in hexadecimal, of tasks awaiting access to the data protected by the region.

Field	Meaning
dddd	Hexadecimal token for the task that currently has access.
eeee	Hexadecimal token for the job that contains the region.
ffffffff	Manner in which waiting tasks are queued at the region. Possible values are FIFO, PRIORITY, and INVALID.

INSPECTING A SEGMENT -- IG

This command displays the principal attributes of the specified segment. The syntax for the IG command is as follows:



PARAMETER

ITEM	Token for the segment to be inspected.

DESCRIPTION

The IG command displays the principal attributes of the segment whose token is represented by ITEM. Figure 4-7 depicts the form of the display produced by IG.

```
                 ----- iRMX86 SEGMENT REPORT -----
SEGMENT TOKEN          bbbb             CONTAINING JOB         dddd
SEGMENT BASE           cccc             SEGMENT LENGTH        eeeee

NAME(S)  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
         aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
```

Figure 4-7. An iRMX™ 86 Segment Report

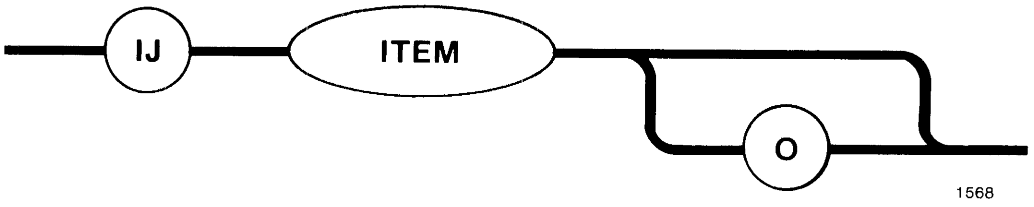
The following describes the fields in Figure 4-7:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the segment is cataloged in the object directory of either the segment's containing job or the root job. If the segment is not cataloged in either directory, "NONE FOUND" is displayed here.
bbbb	Hexadecimal token for the segment.

Field	Meaning
cccc	Base address of the segment.
dddd	Hexadecimal token for the job that contains the segment.
eeeee	Number, in hexadecimal, of bytes in the segment.

INSPECTING A JOB -- IJ

This command lists the principal attributes of a specified job. The syntax for the IJ command is as follows:



PARAMETERS

ITEM	A token for the job to be inspected.
O	If this option is included, the job's object directory is also listed. If omitted, the object directory is not listed.

DESCRIPTION

The IJ command lists the principal attributes of a job whose token is represented by ITEM. It also lists the object directory if the O option is included. If there is a large number of entries in the object directory, the control-O character can be used to prevent data from rolling off the screen. The control-O special character is described in Chapter 2.

Figure 4-8 depicts the form of the display produced by the IJ command.

```
                     ----- iRMX86 JOB REPORT -----
JOB TOKEN               bbbb       PARENT JOB                jjjj
POOL MAXIMUM            cccc       POOL MINIMUM              kkkk
CURRENT ALLOCATED       dddd       CURRENT UNALLOCATED       llll
CURRENT # OBJECTS       eeee       CURRENT # TASKS           mmmm
MAXIMUM # OBJECTS       ffff       MAXIMUM # TASKS           nnnn
CURRENT # CHILD JOBS    gggg       DELETION PENDING           ppp
EXCEPTION MODE          hhhh       EXCEPTION HANDLER    qqqq:rrrr
MAXIMUM PRIORITY        iiii

NAME(S)  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
         aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa

                     ----- OBJECT DIRECTORY -----
MAXIMUM SIZE            uuuu        VALID ENTRIES            vvvv
NAME         TOKEN      NAME         TOKEN      NAME         TOKEN
ssssssssssss tttt       ssssssssssss tttt       ssssssssssss tttt
```

Figure 4-8. An iRMX™ 86 Job Report

The following describes the fields in Figure 4-8:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the job is cataloged in the object directory of either the job's parent job or the root job. If the job is not cataloged in either directory, "NONE FOUND" is printed here.
bbbb	Hexadecimal token for the job.
cccc	Maximum number, in hexadecimal, of 16-byte paragraphs that the job's pool can contain.
dddd	Number of paragraphs that have been either allocated to tasks in the job or lent to child jobs.
eeee	Number, in hexadecimal, of existing objects in job bbbb.
ffff	Maximum number, in hexadecimal, of objects that can exist simultaneously in job bbbb.
gggg	Number, in hexadecimal, of existing jobs that are offspring of job bbbb.

Field	Meaning
hhhh	Exception mode for the job's default exception handler. Possible values are as follows:
iiii	Hexadecimal value that indicates the maximum (numerically lowest) allowable priority for tasks in the job.
jjjj	Hexadecimal token for the parent of job bbbb. If job bbbb is the root job, however, jjjj is "ROOT".
kkkk	Minimum number, in hexadecimal, of 16-byte paragraphs that the job's pool can contain.
llll	Number, in hexadecimal, of unused 16-byte paragraphs in the job's initial pool.
mmmm	Number, in hexadecimal, of tasks currently in the job.
nnnn	Maximum number, in hexadecimal, of tasks that can exist simultaneously in job bbbb.
ppp	Indicator which tells whether a task has attempted to delete the job but was unsuccessful because the job has obtained protection from the DISABLE$DELETION system call. The possible values of ppp are YES and NO.
qqqq	Base, in hexadecimal, of the start address of the job's default exception handler.
rrrr	Hexadecimal offset, relative to qqqq, of the start address of the job's default exception handler.
ssssssssssss	Each such field contains the name under which an object is cataloged in the job's object directory. If there are no entries in the object directory, these fields are blank.
tttt	Each such field contains a token, in hexadecimal, of the object whose name (in the directory) appears next to it.

Values for hhhh:

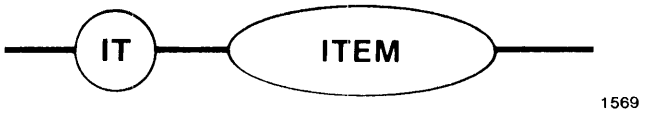
Value	When to Pass Control To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions
INVALID	Never

INSPECT COMMANDS

Field	Meaning
uuuu	Maximum allowable number, in hexadecimal, of entries in the job's object directory.
vvvv	Number, in hexadecimal, of entries currently in the job's object directory.

INSPECTING A TASK -- IT

This command lists the principal attributes of a specified task. The syntax for the IT command is as follows:



PARAMETER

ITEM	Token for the task to be inspected.

DESCRIPTION

The IT command displays the principal attributes of the task whose token is represented by ITEM. Figure 4-9 depicts the form of display produced by IT.

```
                     ----- iRMX86 TASK REPORT -----
  TASK TOKEN                 bbbb     CONTAINING JOB              kkkk
  STACK SEGMENT BASE         cccc     STACK SEGMENT OFFSET        llll
  STACK SEGMENT SIZE         dddd     STACK SEGMENT LEFT          mmmm
  CODE SEGMENT BASE          eeee     DATA SEGMENT BASE           nnnn
  INSTRUCTION POINTER        ffff     TASK STATE              pppppppp
  STATIC PRIORITY            gggg     DYNAMIC PRIORITY            qqqq
  SUSPENSION DEPTH           hhhh     SLEEP UNITS REQUESTED       rrrr
  EXCEPTION MODE             iiii     EXCEPTION HANDLER      ssss:tttt
  NPX TASK                   jjj
  NAME(S)  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
           aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
```

Figure 4-9. An iRMX™ 86 Task Report

The following describes the fields in Figure 4-9:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the task is cataloged in the object directory of either the task's containing job or the root job. If the job is not cataloged in either directory, "NONE FOUND" is displayed here.

Field	Meaning
bbbb	Hexadecimal token for the task.
cccc	Base address, in hexadecimal, of the task's stack segment.
dddd	Size, in bytes, of the task's stack segment.
eeee	Base address, in hexadecimal, of the task's code segment.
ffff	Current value, in hexadecimal, of the task's instruction pointer.
gggg	Hexadecimal priority of the task.
hhhh	Current number, in hexadecimal, of "suspends" against the task. Before the task can be made ready, each "suspend" must be countered with a "resume".
iiii	Exception mode for the task's exception handler. Possible values are as follows:

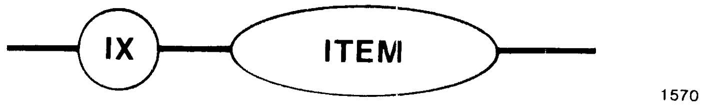
Value	When to Pass Control To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

Field	Meaning
jjj	Indicator which tells whether the task uses the NPX. The possible values of jjj are YES and NO.
kkkk	Hexadecimal token for the task's containing job.
llll	Hexadecimal offset, relative to cccc, of the task's stack segment.
mmmm	Hexadecimal number of bytes currently available in the task's stack.
nnnn	Base address, in hexadecimal, of the task's data segment.
pppppppp	Current execution state of the task. Possible values are "READY", "ASLEEP", "SUSPENDED", "ASLEEP/SUSP", "BROKEN", and "INVALID".
qqqq	A temporary, hexadecimal priority that is sometimes assigned to the task by the Nucleus. This is done to improve system performance.

Field	Meaning
rrrr	If the task is asleep or asleep/suspended, this is the number of 1/100 second sleep units that the task requested just prior to going to sleep. If the task is ready or suspended, qqqq is 0000.
ssss	Base, in hexadecimal, of the start address of the task's exception handler.
tttt	Hexadecimal offset, relative to ssss, of the start address of the task's exception handler.

INSPECTING AN EXTENSION -- IX

This command displays the principal attributes of the specified extension object. The syntax for the IX command is as follows:



PARAMETER

ITEM	Token for the extension object to be inspected.

DESCRIPTION

The IX command displays the principal attributes of the extension whose token is represented by ITEM. Figure 4-10 depicts the form of the display produced by IX.

```
                ----- iRMX86 EXTENSION REPORT -----
     EXTENSION TOKEN         bbbb          CONTAINING JOB       dddd
     TYPE CODE               cccc          DELETION MAILBOX     eeee

     NAME(S)  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
             aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa  aaaaaaaaaaaa
```

Figure 4-10. An iRMX™ 86 Extension Report

The following describes the fields in Figure 4-10:

Field	Meaning
aaaaaaaaaaaa	Each such field contains a name under which the extension is cataloged in the object directory of either the job containing the extension or the root job. If the extension is not cataloged in either directory, "NONE FOUND" is displayed here.
bbbb	Hexadecimal token for the extension.

Field	Meaning
cccc	Hexadecimal type code associated with composite objects licensed by this extension.
dddd	Hexadecimal token for the job containing this extension.
eeee	Hexadecimal token for the deletion mailbox associated with the extension. If there is no deletion mailbox for the extension, "NONE" is displayed here.

COMMANDS TO VIEW OBJECT LISTS

The commands in this section allow you to view lists of iRMX 86 objects. Figure 4-11 illustrates the general syntax for commands in this section.

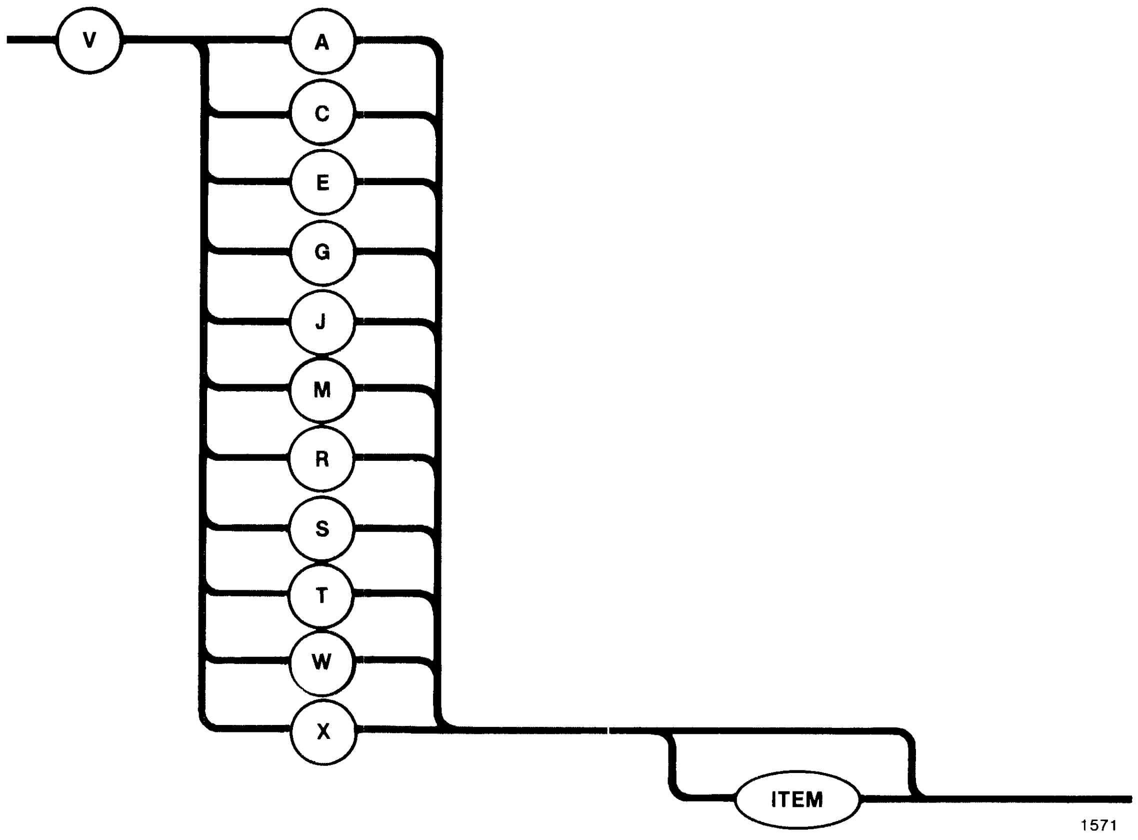


Figure 4-11. Syntax Diagram For Viewing iRMX™ 86 Object Lists

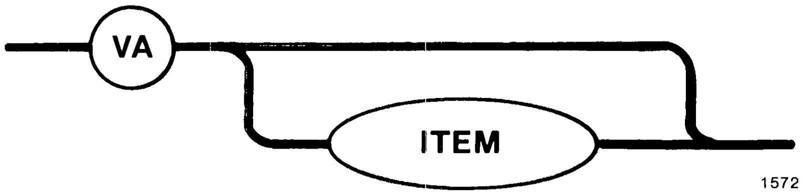
The second letter of the command indicates the type of object list to display, as follows:

J	Jobs
T	Tasks
R	Ready tasks
S	Suspended tasks
A	Asleep tasks
E	Exchanges
W	Waiting Task queues
M	Mailbox queues
G	Segments
C	Composites
X	Extensions

The remainder of this section describes the commands in detail.

VIEWING THE ASLEEP TASKS -- VA

This command displays a list of asleep tasks. The syntax for the VA command is as follows:



PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists only those asleep tasks that are contained in the specified job. If this option is omitted, all asleep tasks in the system are listed.

DESCRIPTION

The VA command displays suspended tasks as:

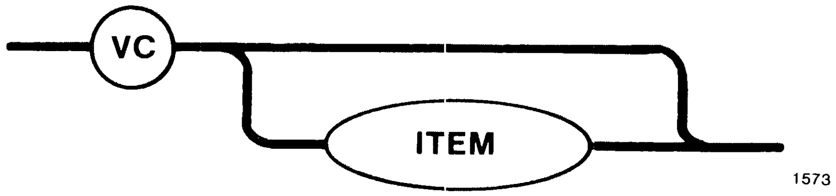
```
    SA = jjjjJ/ttttT  jjjjJ/ttttT ... jjjjJ/ttttT
```

where:

tttt	Token of an asleep task.
jjjj	Token for the job containing the task. An asterisk following the task token indicates that the task has overflowed its stack.

VIEWING COMPOSITES -- VC

This command displays a list of composite objects. The syntax for the VC command is as follows:



PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists only the composite objects contained in the specified job. If this option is omitted, all composite objects in the system are displayed.

DESCRIPTION

The VC command displays composite objects as:

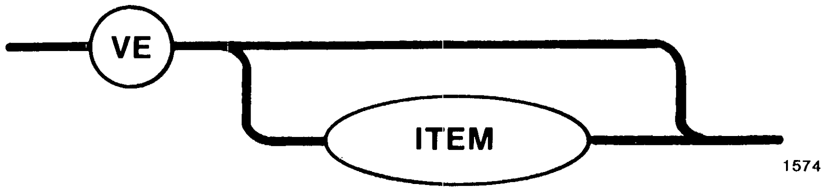
```
    CL = jjjjJ/ccccC jjjjJ/ccccC ... jjjjJ/ccccC
```

where:

cccc	Token for a composite object.
jjjj	Token for the job containing the composite object.

VIEWING EXCHANGES -- VE

This command displays a list of exchanges. The syntax for the VE command is as follows:



PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists only those exchanges that are contained in the specified job. If this option is omitted, all exchanges in the system are listed.

DESCRIPTION

The VE command lists exchanges as:

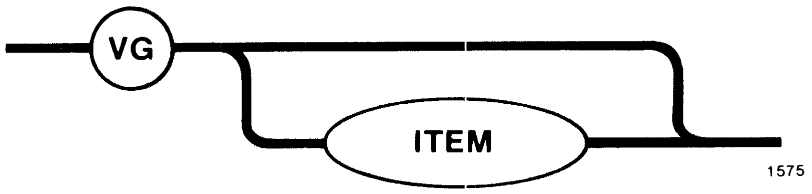
```
    EL = jjjjJ/xxxxt jjjjJ/xxxxt ... jjjjJ/xxxxt
```

where:

xxxx	Token for an exchange.
t	Type of the exchange (M for mailbox, S for semaphore, or R for region).
jjjj	Token for the job containing the exchange.

VIEWING SEGMENTS -- VG

This command displays a list of segments. The syntax for the VG command is as follows:



PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists only the segments contained in the specified job. If this option is omitted, all segments in the system are displayed.

DESCRIPTION

The VG command displays segments as:

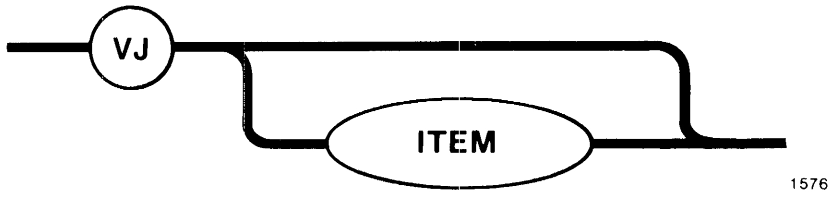
```
    GL = jjjjJ/ggggG jjjjJ/ggggG ... jjjjJ/ggggG
```

where:

gggg	Token for a segment.
jjjj	Token for the job containing the segment.

VIEWING JOBS -- VJ

This command displays a list of jobs. The syntax for the VJ command is as follows:



PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists only those jobs that are children of the specified job. If this option is omitted, all jobs in the system are listed.

DESCRIPTION

The VJ command displays jobs as:

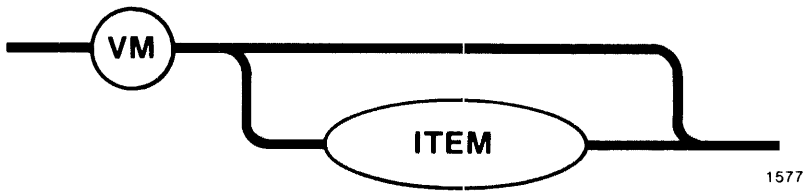
```
    JL = ppppJ/jjjjJ ppppJ/jjjjJ ... ppppJ/jjjjJ
```

where:

jjjj	Job token.
pppp	Token of its parent job. If the job designated by jjjj is the root job, then "ROOT" replaces "ppppJ".

VIEWING MAILBOX OBJECT QUEUES -- VM

This command displays object queues at mailboxes. The syntax for the VM command is as follows:



PARAMETER

ITEM	Token for a mailbox or a job. If you specify a mailbox token for this option, the Debugger lists only the object queue associated with the specified mailbox. If you specify a job token for this option, the Debugger lists all object queues in the specified job. If you omit this option, the Debugger displays object queues for all exchanges in the system.

DESCRIPTION

The VM command displays object queues at mailboxes as:

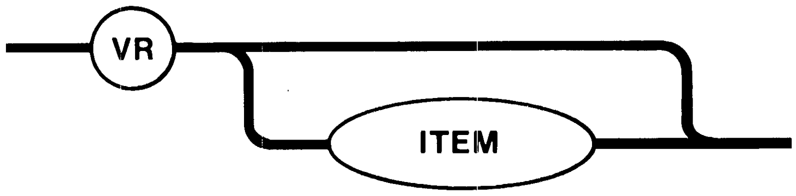
```
    ML jjjjJ/mmmmM = jjjjJ/oooot jjjjJ/oooot ... jjjjJ/oooot
    ML jjjjJ/mmmmM = jjjjJ/oooot jjjjJ/oooot ... jjjjJ/oooot

                           .
                           .
                           .

    ML jjjjJ/mmmmM = jjjjJ/oooot jjjjJ/oooot ... jjjjJ/oooot
```

where:

mmmm	Token for a mailbox.
oooo	Token for an object in that mailbox's object queue.
t	Type of the object (J for job, T for task, M for mailbox, S for semaphore, and G for segment).
jjjj	Token for the job containing the mailbox or object.

VIEWING READY TASKS -- VR

This command displays a list of ready tasks. The syntax for the VR command is as follows:



1578

PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists, in priority order, the ready tasks that are contained in the specified job. If this option is omitted, all ready tasks in the system are listed in order of priority.

DESCRIPTION

The VR command displays ready tasks as:

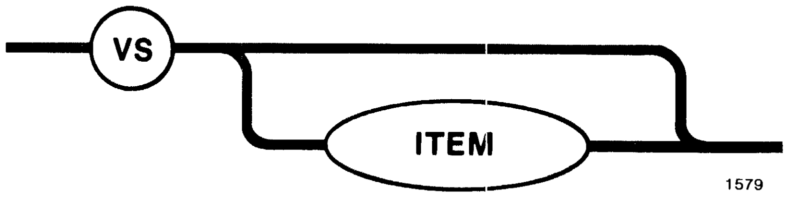
```
    RL = jjjjJ/ttttT jjjjJ/ttttT ... jjjjJ/ttttT
```

where:

tttt	Token of a ready task.
jjjj	Token for the job containing the task. An asterisk following a task token indicates that the task has overflowed its stack.

VIEWING SUSPENDED TASKS -- VS

This command displays a list of suspended tasks. The syntax for the VS command is as follows:



PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists only those suspended tasks that are contained in the specified job. If this option is omitted, all suspended tasks in the system are listed.

DESCRIPTION

The VS command displays suspended tasks as:

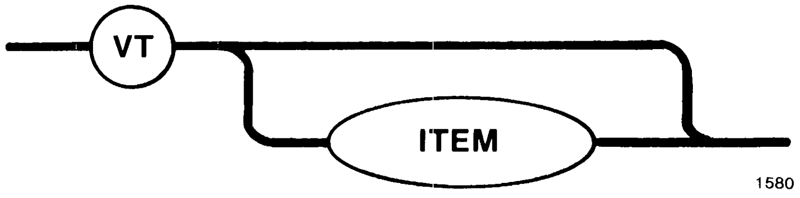
```
    SL = jjjjJ/ttttT jjjjJ/ttttT ... jjjjJ/ttttT
```

where:

tttt	Token of a suspended task.
jjjj	Token for the job containing the task. An asterisk following a task token indicates that the task has overflowed its stack.

VIEWING TASKS -- VT

This command displays a list of tasks. The syntax for the VT command is as follows:



PARAMETER

ITEM	Token for a job. If this option is included, the Debugger lists only those tasks that are contained in the specified job. If this option is omitted, all tasks in the system are listed.

DESCRIPTION

The VT command displays tasks as:

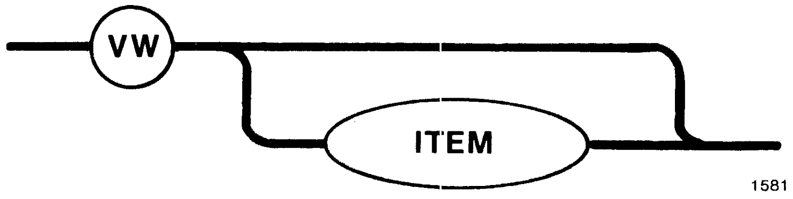
```
     TL = jjjjJ/ttttT jjjjJ/ttttT ... jjjjJ/ttttT
```

where:

tttt	Task token.
jjjj	Token for the job that contains the task. An asterisk following a task token indicates that the task has overflowed its stack.

VIEWING WAITING TASK QUEUES -- VW

This command displays the waiting task queues at exchanges. The syntax for the VW command is as follows:



PARAMETER

ITEM	Token for an exchange or a job. If you specify an exchange token for this option, the Debugger lists only the task queue associated with the specified exchange. If you specify a job token for this option, the Debugger lists all task queues in the specified job. If you omit this option, the Debugger displays task queues for all exchanges in the system.

DESCRIPTION

The VW command displays task queues at exchanges as:

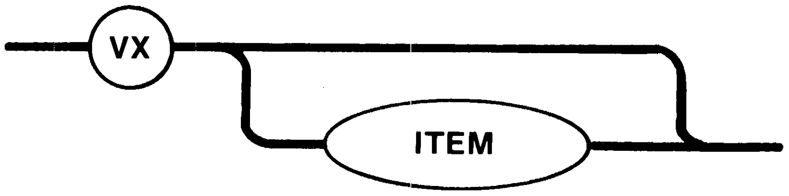
```
    WL jjjjJ/xxxxt = jjjjJ/ttttT jjjjJ/ttttT ... jjjjJ/ttttT
    WL jjjjJ/xxxxt = jjjjJ/ttttT jjjjJ/ttttT ... jjjjJ/ttttT

                     .
                     .
                     .

    WL jjjjJ/xxxxt = jjjjJ/ttttT jjjjJ/ttttT ... jjjjJ/ttttT
```

where:

xxxx	Token for an exchange.
t	Type of the exchange (M for mailbox, S for semaphore, or R for region).
tttt	Token for a task which is queued at that exchange.
jjjj	Token for the job containing the task. An asterisk indicates that either the task has overflowed its stack or the task was loaded by the Application Loader.

VIEWING EXTENSIONS -- VX

This command displays either a list of extension objects or a list of composite objects associated with a particular extension object. The syntax for the VX command is as follows:



1582

PARAMETER

ITEM	Token for an extension object. If this option is included, the Debugger lists all composite objects associated with the specified extension object. If this object is omitted, the Debugger lists all extension objects in the system.

DESCRIPTION

If the ITEM parameter is omitted, the VX command displays extension objects as follows:

```
    XL = jjjjJ/xxxxX jjjjJ/xxxxX ... jjjjJ/xxxxX
```

where:

xxxx	Token for an extension object.
jjjj	Token for the job containing the extension.

If the ITEM option is included, the VX command lists the composite objects associated with a particular extension object as follows:

```
    XL jjjjJ/xxxxX = kkkkJ/ccccC kkkkJ/ccccC ... kkkkJ/ccccC
```

where:

xxxx	Token for the extension object.

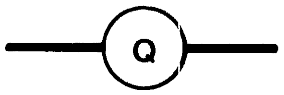
jjjj	Token for the job containing the extension.
cccc	Token for the composite object that is associated with the specified extension.
kkkk	Token for the job containing the composite object.

COMMANDS TO EXIT THE DEBUGGER

The Q command described in this section allows you to exit the Debugger and resume processing.

EXITING THE DEBUGGER -- Q

This command exits the Debugger. The syntax for the Q command is as follows:



1583

DESCRIPTION

The Q command deactivates the Debugger. When a debugging session is terminated, the tables and lists the Debugger maintains are not destroyed. Q also displays the message "EXIT iRMX 86 DEBUGGER".

***

CHAPTER 5 CONFIGURATION

The Debugger is a configurable layer of the Operating System. It contains several options that you can adjust to meet your specific needs. To make configuration choices, Intel provides three kinds of information:

- A list of configurable options.
- Detailed information about the options.
- Procedures to allow you to specify your choices.

The balance of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the iRMX 86 CONFIGURATION GUIDE.

Debugger configuration is almost identical to Terminal Handler configuration (except that only one Debugger can be present in the application system). Debugger configuration involves selecting characteristics of the Debugger's Terminal Handler and specifying information about the processor board and the terminal. The following sections describe the configurable options available on the Debugger.

BAUD RATE

You can set the baud rate for the Debugger's Terminal Handler to any of the following values:

```
  110
  150
  300
  600
 1200
 2400
 4800
 9600
19200
```

The default baud rate for the Debugger's Terminal Handler is 9600.

BAUD COUNT

The baud count provides a way to calculate internal timer values given the clock input frequency. The baud count sets the limits on the baud rate attributes of the Debugger's Terminal Handler. If your system's programmable interval timer (PIT) has a clock input frequency other than 1.2288 megahertz, you must set the baud count. The default value for the baud count is 4.

RUBOUT MODE AND BLANKING CHARACTER

There are two ways to rubout a character:

- Copying mode
- Blanking mode

In the copying mode, the character being deleted from the current line is re-echoed to the display. For example, entering "CAT" and then striking RUBOUT three times results in the display "CATTAC".

In the blanking mode, the deleted character is replaced on the CRT screen with the blanking character. For example, entering "CAT" and then striking RUBOUT three times deletes all three characters from the display.

The copy mode is the default mode. The default blanking character for the blanking mode is a space (20H).

USART

The USART is a device that, depending upon the application, can be used either to convert serial data to parallel data or to convert parallel data to serial data. The Debugger's Terminal Handler requires a 8251A USART as a terminal controller. You can specify:

- The port address of the USART. The default value for the port address is 0D8H.
- The interval between the port addresses for the USART.
- The number of bits of valid data per character that can be sent from the USART. The default value for the number of bits is 7.

PIT

You can specify the following information about the programmable interval timer (PIT):

- The port address of the PIT. The default value for the port address is 0D0H.
- The interval between the port addresses for the PIT.
- The number of the PIT counter connected to the USART clock input. The default value is 2.

MAILBOX NAMES

You can change the default names of both the input mailbox (RQTHNORMIN) and the output mailbox (RQTHNORMOUT). The new names must not be over 12 alphanumeric characters in length.

INTERRUPT LEVELS

You can specify the interrupt levels used by the Debugger's Terminal Handler for input and output. You choose interrupt levels by selecting a value that corresponds to a particular interrupt value. The default value for the input interrupt level is 68H and the default value for the output interrupt level is 78H.

***

APPENDIX A ERROR MESSAGES

This appendix lists the error messages that can occur when you enter Debugger commands. Since the Debugger reads commands on a line-by-line basis, it will not issue an error message for a command until you terminate the command with an end-of-line character (carriage return or line feed). Then, if the Debugger detects an error, it generates a display of the following form:

```
command portion #
error message
```

where command portion consists of the command up to the point where the Debugger detected the error, and error message consists of one of the following:

Message	Description
ATTEMPT TO MODIFY NON-RAM LOCATION	You tried to define a breakpoint at a non-RAM memory location.
BREAKPOINT TASK NOT AN NDP TASK	You specified the N command, but the breakpoint task was not designated as an Numeric Processor Extension task at its creation.
COMMAND TOO COMPLEX	In order to process your commands, the Debugger maintains a semantic stack, on which it places all the semantic entities of your command. Your command was too complex and overflowed this stack. To correct this problem, you should first define numeric variables for some of your more complex expressions, and then use these variables in your command in place of the expressions.
DEBUGGER POOL TOO SMALL	In order to process your command, the Debugger tried to create an iRMX 86 segment. However, there was not enough free space in the system to create this segment.
DUPLICATE SYMBOL	You attempted to define a numeric or breakpoint variable name that was already defined.

Message	Description
EXECUTION BREAKPOINT ALREADY DEFINED	You attempted to define (or redefine) an execution breakpoint at an address which already specifies an execution breakpoint. This breakpoint may have been set up by the Debugger or by the iSBC 957B Monitor and must be deleted before a new one can use this location.
INTERRUPT TASK NOT ON BREAKPOINT LIST	You attempted to make an interrupt task the current breakpoint task without first suspending that interrupt task. An interrupt task can only be made the current breakpoint task by first incurring a breakpoint.
INVALID TASK STATE	The Nucleus-maintained task descriptor contains inconsistent information. You have probably overwritten this area of memory. It is unlikely that the task can continue to run.
INVALID TOKEN	You specified a token for a different kind of object than that required by the command.
ITEM NOT FOUND	You tried to delete or change a nonexistent numeric variable.
NO BREAKPOINT TASK	You entered the R or N command without first establishing a breakpoint task.
SYNTAX ERROR	The command is syntactically incorrect.
TASK NOT ON BREAKPOINT LIST	You tried to remove a task from the breakpoint list with the G command when the task was not on the list.
TASK NOT SUSPENDABLE. WILL BE BROKEN WHEN SUSPENDABLE	You entered the BT command to establish a breakpoint task, but the Debugger could not suspend the task in its current state (for example, the task currently has access to a region). The Debugger will suspend the task when it becomes possible to do this.
UNDEFINED SYMBOL	The Debugger was unable to find the specified symbol in the local symbol table, the object directory of the breakpoint task's job, or the root object directory.

UNKNOWN BREAKPOINT iAPX 86, 88 MONITOR NOT CONFIGURED	The Debugger encountered a breakpoint for which it had no record. It tried to pass the breakpoint to the Monitor but could not because the Monitor is not included in your system.

***

INDEX

Primary references are underscored.

***

iRMX™ 86 SYSTEM DEBUGGER REFERENCE MANUAL

CONTENTS

PAGE

FIGURES

PAGE

***

CHAPTER 1
ORGANIZATION

This manual contains four chapters. Some of the chapters contain introductory or overview material that you might not need to read if you are already familiar with the iSBC 957B, iSDM 86, or iSDM 286 monitor. Other chapters contain reference material that you can use as you debug your system. You can use this chapter to determine which of the other chapters you should read.

The remaining chapters of the manual are the following:

Chapter 2	This chapter describes the features of the System Debugger and its relationship to the other tools for debugging iRMX 86 applications. You should read this chapter if you are going through the manual for the first time.
Chapter 3	This chapter gives a variety of facts pertaining to the use of the System Debugger. You should read this chapter if you are installing the System Debugger and/or configuring it into your system.
Chapter 4	This chapter contains detailed descriptions of the System Debugger commands. The commands are listed in alphabetical order for easy referencing. When you are debugging your system you should refer to this chapter for specific information about the format and parameters of the commands.

***

CHAPTER 2 INTRODUCTION

The development of almost every system requires debugging. To aid you in the development of iRMX 86-based application systems, Intel provides the iRMX 86 Debugger, the ICE-86 In-Circuit Emulator, the iSDM 86 and iSDM 286 System Debug Monitors, and the iSBC 957B monitor. The System Debugger extends the capabilities of the three monitors. This manual describes the System Debugger extension. The iRMX 86 DEBUGGER REFERENCE MANUAL describes the iRMX 86 Debugger. The USER'S GUIDE FOR THE iSBC 957B iAPX 86, 88 INTERFACE AND EXECUTION PACKAGE describes the iSBC 957B monitor. The iSDM 86 SYSTEM DEBUG MONITOR REFERENCE MANUAL describes the iSDM 86 monitor. And the iSDM 286 SYSTEM DEBUG MONITOR REFERENCE MANUAL describes the iSDM 286 monitor. The following sections describe the relative advantages of the various debugging tools.

ADVANTAGES OF THE iRMX™ 86 DEBUGGER

The iRMX 86 Debugger is a debugging tool that is "sensitive" to the data structures that the Nucleus maintains. The iRMX 86 Debugger allows you to:

- Manipulate or examine any task while other tasks in the system continue to run. This distinguishes the iRMX 86 Debugger from the iRMX 86 System Debugger, which requires that the application system be "frozen."

- Monitor system activity without interfering with execution.

- Examine and interpret data structures that are associated with the Nucleus and the Nucleus objects.

ADVANTAGES OF THE ICE®-86 EMULATOR

The ICE-86 emulator provides in-circuit emulation for iAPX 86, 88 microprocessor-based systems, meaning that it "stands in" for the 8086 or 8088 microprocessor in your target iRMX 86-based system during development. The ICE-86 emulator allows you to:

- Get closer to the hardware level by examining the contents of input pins and input ports.

- Change the values at output ports.

- Examine individual components rather than an entire board.

- Look at the most recent 80 to 150 assembly language instructions executed.

ADVANTAGES OF THE iSBC® 957B, iSDM™ 86, AND iSDM™ 286 MONITORS

The iSBC 957B, iSDM 86, and iSDM 286 monitors each support both interactive commands and system I/O routines. Each allows you to:

- Disassemble code.
- Set execution and memory breakpoints.
- Display memory.

ADVANTAGES OF THE iRMX™ 86 SYSTEM DEBUGGER

You can extend the capabilities of the iSBC 957B, iSDM 86, or iSDM 286 monitor the System Debugger part of your operating system. In addition to retaining the features of the monitors, the System Debugger:

- Identifies and interprets iRMX 86 system calls.
- Displays iRMX 86 objects.
- Examines the stack of a task to determine which iRMX 86 system calls it has made recently.

REQUIREMENTS OF THE iRMX™ 86 SYSTEM DEBUGGER

In order to use the System Debugger, you must have exactly one of the following hardware configurations, with whatever support hardware that is required (independent of the System Debugger):

- A terminal connected directly to an iAPX 86-, 88-, 186-, 188-, or 286-based board.

or

- An Intellec system connected to an iAPX 86-, 88-, 186-, 188-, or 286-based board.

You must also have:

- The monitor portion of the iSBC 957B iAPX 86, 88 Interface and Execution Package or the iSDM 86 or iSDM 286 System Debug Monitor.
- At least the minimal configuration of the Nucleus. The System Debugger needs only a small portion of valid Nucleus code, so most of the System Debugger commands will function even if you accidentally write over part of the Nucleus.

See the next chapter for more information about configuring and installing the System Debugger.

***

CHAPTER 3
USING THE SYSTEM DEBUGGER

This chapter contains various facts about using the iRMX 86 System Debugger.

HOW THE SYSTEM DEBUGGER IS SUPPLIED

The System Debugger is supplied as a file along with the other parts of the iRMX 86 Operating System.

USE RESTRICTIONS OF THE SYSTEM DEBUGGER

One of the capabilities of the System Debugger is that it can display information about specific invocations of system calls. However, it can do this correctly only for applications that use the PL/M-86 SMALL, COMPACT, or LARGE model of segmentation.

CONFIGURING THE SYSTEM DEBUGGER

To use the System Debugger to debug your application, you must configure it into the application. You do this simply by responding to two prompts that the iRMX 86 Interactive Configuration Utility issues. One of the prompts asks whether you want the System Debugger to be part of your system. The other, which applies only if you respond affirmatively to the first prompt, asks which interrupt level you want to use to invoke the System Debugger manually.

INVOKING THE SYSTEM DEBUGGER

There are two ways of invoking the System Debugger. As the previous section implies, one way is to press the button that is physically tied to the interrupt level you specify during configuration. The other way, which requires that your system include the Human Interface, is to use the DEBUG command.

The DEBUG command syntax requires the pathname of a loadable file. DEBUG loads the indicated file and then passes control to the iSBC 957B, iSDM 86, or iSDM 286 monitor. Normally, the file the DEBUG command loads is the file that is to be debugged. However, in this case the file to be debugged (the application system incorporating the System Debugger) is already in memory. To satisfy the requirement that the DEBUG command load some file, but without corrupting your application, specify the pathname of a file that the DEBUG command can load harmlessly into an area of memory not used by the application. A file you can use for this purpose is the TIME command of the Human Interface. It requires little memory and, when loaded, is automatically located where it does not interfere with the application.

See the iRMX 86 OPERATOR'S MANUAL for more information concerning the DEBUG command.

After the DEBUG command loads the file into memory or after you press the interrupt button, the monitor issues its period (.) prompt, and you can begin entering System Debugger commands. These commands are the subject of the next chapter.

RETURNING TO YOUR APPLICATION

When you have finished debugging your application system, you can start it up again by means of the go (G) command of the monitor.

***

CHAPTER 4 COMMANDS

This chapter contains detailed descriptions of the iRMX 86 System Debugger commands, in alphabetical order. There is also a Command Dictionary that lists the commands in functional groups.

This chapter uses "CS:IP" to mean "code segment:instruction pointer." The chapter also contains several examples of System Debugger commands entered at the terminal. In the examples, user input is underscored to distinguish it from System Debugger output. Carriage returns are not shown after the user input but they are required for the System Debugger to execute the command.

CHECKING VALIDITY OF TOKENS

The iRMX 86 Operating System maintains tokens in doubly-linked lists. Whenever you enter a command that requires a token as a parameter, the System Debugger checks the validity of that token by looking at the token's forward and backward links. It checks tokens that you enter as parameters for the VD, VK, VJ, VO, VR, VT, and VU commands as well as the tokens that are listed in the displays.

If one of a token's links is bad, the System Debugger generates an error message along with the information the command that you entered usually displays. The token you enter as a parameter of the System Debugger command always appears in each line as the center value in the display of tokens. The displays for forward- and backward-link errors are as follows:

```
Forward link ERROR:   4111-->4E85    4111<--4E85-->4155   ?FFFF<--4155

   Back link ERROR:   4111-->410F?   4111<--4E85-->4155   4E85<--4155
```

Arrows to the right indicate forward links and those to the left indicate backward links. A question mark appearing before or after a value signals a forward or backward link error.

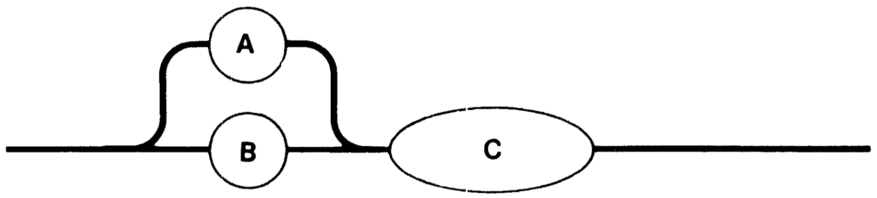
If both links are bad, the System Debugger considers the token invalid and displays the following message:

```
*** INVALID TOKEN ***
```

The presence of a link error means that iRMX 86 data structures have been corrupted. The most common reason for this problem is overwriting. One of your tasks might have accidentally written over part of the system data structures and/or code. If you are using the non-maskable interrupt, another possible cause of a link error is that you interrupted the Nucleus while it was setting up the links. If either of these things happen, you must re-initialize the System Debugger (and perhaps your System). Only then can you use the VD, VJ, VK, VO, VR, VT, and VU commands without getting another link error. See Chapter 3 in this manual for more information about initializing the System Debugger on your system.

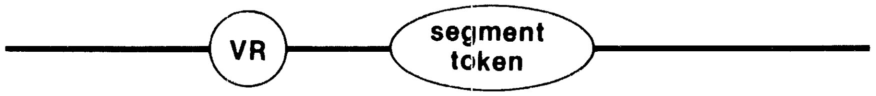
PICTORIAL REPRESENTATION OF SYNTAX

This manual uses a schematic device to illustrate the syntax of commands. The schematic consists of what looks like an aerial view of a model railroad, with syntactic elements scattered along the track. Imagine that a train enters the system at the far left, drives around as much as it can or wants to (sharp turns and backing up are not allowed), and finally departs at the far right. The command it generates in doing so consists of the syntactic elements that it encounters on its journey. The following pictorial syntax shows two valid sequences: AC and BC.



x-455

The pictorial syntax of the commands in this chapter does not show spaces as elements. However, the SDB does allow one or more spaces between the command and the parameter. For example, even though the syntax for VR is:



you can enter:

x-456

.<u>VR xxxx</u>

The space between "VR" and "xxxx" does not affect the result of the command.

Even though all syntax diagrams show uppercase letters, such as VR, entering lowercase equivalents of those letters produces the same effect.

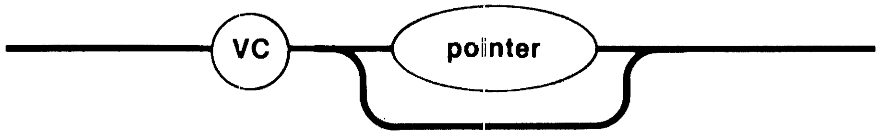
DISPLAY OF NUMERICAL VALUES

In all of the displays that this chapter discusses, all numerical values are given in hexadecimal form.

COMMAND DICTIONARY

VC--Display System Call Information

The VC command checks to see if a CALL instruction is an iRMX 86 system call.



x-457

PARAMETER

pointer	The optional pointer parameter can be any valid iSBC 957B, iSDM 86, or iSDM 286 address. The System Debugger uses this address as the address of the CALL instruction to be checked. If you do not supply a pointer, the System Debug Monitor uses the default pointer, which is the current CS:IP. If you specify an IP value but not a CS value, the System Debugger uses the current CS as the default base.

DESCRIPTION

If the CALL instruction is an iRMX 86 system call, the VC command displays information about the CALL instruction as shown in Figure 4-1.

```
    S/W int:  xx (subsystem)      entry code xxxx     system call
```

Figure 4-1. Format Of VC Output

The fields in Figure 4-1 are defined as follows:

S/W int: xx (subsystem)	The software interrupt number and the iRMX 86 subsystem that corresponds to that number.
entry code xxxx	The entry code for the system call within the subsystem.
system call	The name of the iRMX 86 system call.

NOTE

The System Debugger uses the software interrupt number associated with the displayed entry code to determine whether the CALL instruction represents a system call. It is possible, but not likely, that the System Debugger can interpret a sequence of bytes as a software interrupt (INT) instruction and then (inaccurately) reported that a CALL instruction is an iRMX 86 system call.

ERROR MESSAGES

The System Debugger returns the following error messages for the VC command:

Error Message	Description
Syntax Error	You made an error in entering the command.
Not a system CALL	The parameter you specified points to a CALL instruction that is not an iRMX 86 system call.
Not a CALL instruction	The parameter you specified does not point to any kind of call instruction.

EXAMPLES

Suppose you disassembled the following code using the DX command of the iSBC 957, iSDM 86, or iSDM 286 monitor:

```
49A4:006D 50          PUSH      AX
49A4:006E E8AD1E      CALL      A = 1F1E          ;$+7856
49A4:0071 E8DD03      CALL      A = 0451H         ;$+992
49A4:0074 B80000      MOV       AX,0
49A4:0077 50          PUSH      AX
49A4:0078 8D060600    LEA       AX,WORD PRT 006H
49A4:007C 1E          PUSH      DS
49A4:007D 50          PUSH      AX
49A4:007E E8411E      CALL      A = 1EC2H         ;$+7748
49A4:0081 A30000      MOV       WORD PTR 0000H,AX
```

If you use the VC command on the CALL instruction at address 49A4:006E, that is, you enter:

```
.VC 49A4:006E
```

the System Debugger responds by displaying the following information:

```
    S/W Int:  B8 (Nucleus)   entry code 0801     set exception handler
```

The "S/W Int: B8 (Nucleus)" means that the software interrupt number, "B8", identifies this call as a Nucleus call. The entry code within the Nucleus is "0801" which corresponds to an RQ$SET$EXCEPTION$HANDLER system call.

Now suppose you want to see if the CALL instruction at 49A4:0071 is a system call. Enter:

```
    .VC 49A4:0071
```

The System Debugger responds with the following message.

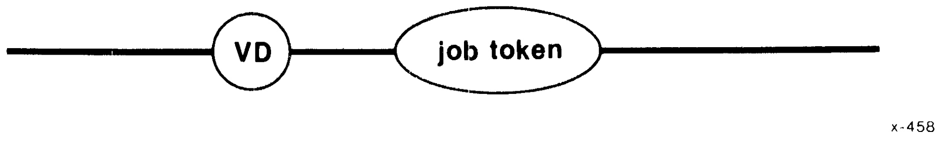
```
    Not a system CALL
```

Finally, if you use the VC command on the instruction at 49A4:0074, the System Debugger responds with:

```
    Not a CALL instruction
```

VD--Display A Job's Object Directory

The VD command displays a job's object directory.



PARAMETER

job token	The token for the job whose object directory you want to display.

DESCRIPTION

If the parameter is a valid job token, the System Debugger displays the job's object directory, as shown in Figure 4-2.

```
Directory size:    xxxx           Entries used:    xxxx

name_1             token_1
name_2             tasks waiting  token_2...token_i
 .                  .
 .                  .
 .                  .
name_j             token_j
invalid entry
name_k             token_k
 .                  .
 .                  .
 .                  .
name_n             token_n
```

Figure 4-2. Format Of VD Output

The fields in Figure 4-2 are as follows:

Directory size	The maximum allowable number of entries this job can have in its object directory.
Entries used	The number of entries used presently in the directory.
$name_1...name_n$	The names under which objects are cataloged.
$token_1...token_n$	Tokens for the cataloged objects.
tasks waiting	Signifies that one or more tasks have performed an RQ$LOOKUP$OBJECT on an object that is not cataloged. The tokens following this field identify the tasks that are still waiting for the objects to be cataloged.
invalid entry	This field appears only if the specified job's object directory has been destroyed or written over.

ERROR MESSAGES

The System Debugger returns the following error messages for the VD command:

Error Message	Description
Syntax Error	You did not specify a parameter for the command, or you made an error in entering the command.
TOKEN is not a Job	You entered a valid token that is not a job token.
*** INVALID TOKEN ***	The value you entered for the token is not a valid token.

EXAMPLES

If you want to look at the object directory of job "528F," you can enter:

```
    .VD 528F
```

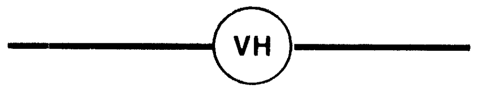
The System Debugger responds as follows:

```
    Directory size:  000A          Entries used:  0003

    $               5229
    R?IOUSER        5201
    RQGLOBAL        528F
```

The symbols "$," "R?IOUSER," and "RQGLOBAL" are the names of the objects, and their respective tokens are 5229, 5201, and 528F. There are no waiting tasks or invalid entries.

VH--Display Help Information

The VH command displays and describes the ten System Debugger commands.



x-459

PARAMETERS

There are no parameters for this call.

DESCRIPTION

The VH command lists all of the System Debugger commands, along with their parameters and a short description of each command.

ERROR MESSAGE

The System Debugger returns the following error message for the VH command:

Error Message	Description
Syntax Error	You made an error in entering the command.

EXAMPLE

If you enter:

```
.VH
```

the System Debugger responds as shown in Figure 4-3, where the brackets indicate optional parameters.

```
iRMX 86 SYSTEM DEBUGGER, Vx.y

   vc [<POINTER>]       Display system call,
   vd <Job TOKEN>       Display job's object directory.
   vh                   Display help information.
   vj [<Job TOKEN>]     Display job hierarchy from specified level.
   vk                   Display ready and sleeping tasks.
   vo <Job TOKEN>       Display list of objects for specified job.
   vr <Seg TOKEN>       Display I/O Request/Result Segment.
   vs [<Count>]         Display stack and system call information.
   vt <TOKEN>           Display iRMX 86 object.
   vu <Task TOKEN>      Display system calls on stack of specified task.
```

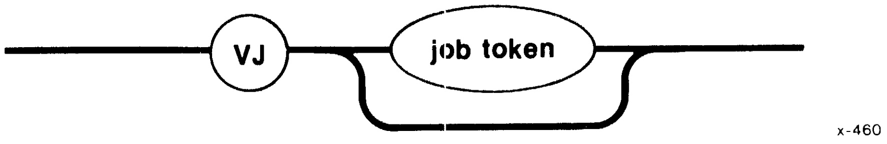
Figure 4-3. VH Display

NOTE

If you use zero (0) for any of the optional parameters shown in Figure 4-3, the effect is the same as if you omitted the parameter altogether.

VJ--Display The Job Hierarchy

The VJ command displays the portion of the job hierarchy that descends from the level you specify.



PARAMETER

job token	The token for the job whose descendant job hierarchy you want to display.
	If you do not specify a job token, VJ assumes the default job, which is the root job.
	The specified job, whether it is specified explicitly or whether it is the default (root) job, should not have more than 44 generations of job descendants. Otherwise, the display of the excessively-long branch is discontinued, an error message is displayed, and the System Debugger prompts for another command.

DESCRIPTION

The VJ command displays the token of the specified job and all the tokens of its descendant jobs. It also displays the tokens of the jobs (and their descendants) at the same level as the specified job. The descendant jobs are indented three spaces to show their position in the hierarchy. This command displays the job hierarchy as shown in Figure 4-4.

```
      iRMX/86 Job Tree

token1
   token2
      token3
         token4
      token5
   token6
```

Figure 4-4. Format Of VJ Output

The fields in Figure 4-4 are as follows:

$token_1$	The token for the root job or the job you specify.
$token_2 \ldots token_6$	The tokens for the descendant jobs of the root job or the job you specify.

In Figure 4-4, jobs 2 and 6 are both indented three spaces to signify that they are children of job 1. Similarly, jobs 3 and 5 are depicted as children of job 2, and job 4 is shown as the child of job 3.

ERROR MESSAGES

The System Debugger returns the following error messages for the VJ command:

Error Message	Description
Syntax Error	You made an error in entering the command.
TOKEN is not a Job	You entered a valid token that is not a job token.
*** INVALID TOKEN ***	The value you entered for the token is not a valid token.
Error looking for root job	The System Debugger cannot find the root job.
SDB job nest limit exceeded	The job specified in the command invocation (or the default job) has more than 44 generations of job descendants.

EXAMPLES

If you want to examine the hierarchy of the root job, enter:

```
.VJ
```

Suppose the System Debugger responds with the following job tree.

```
               iRMX/86 Job Tree

    57DE
       528F
          51CE
             4F9F
       5741
       57B5
```

The display shows "57DE" to be the root job.

If you want to display the descendant jobs of "51CE", enter:

```
    .VJ 51CE
```

The System Debugger displays the following job tokens:

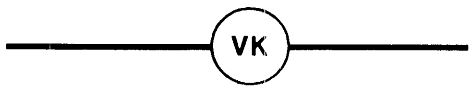
```
          51CE
             4F9F
```

NOTE

The VJ command (without a parameter) requires the Nucleus interrupt vector and a small part of the Nucleus code in order to function correctly. If you destroy the Nucleus interrupt vector (by pressing the RESET switch) or if you write over the required part of Nucleus code, this command does not operate properly. You must re-initialize your system in order to restore the VJ command.

VK--Display Ready And Sleeping Tasks

The VK command displays the tokens for the tasks that are in the ready and sleeping states.



x-461

PARAMETERS

This command has no parameters.

DESCRIPTION

The VK command displays the tokens for the tasks that are ready and asleep, in the format shown in Figure 4-5.

```
Ready tasks:    xxxx
Sleeping tasks: xxxx
```

Figure 4-5. Format Of VK Output

The fields in Figure 4-5 are as follows:

Ready tasks	The tokens for all ready tasks in the system.
Sleeping tasks	The tokens for all sleeping tasks in the system.

ERROR MESSAGES

The System Debugger returns the following error messages for the VK command:

Error Messages	Description
Ready tasks: Can't locate Sleeping tasks: Can't locate	The system is corrupted. See the following explanation.
Syntax error	You made an error in entering the command.

The System Debugger uses the Nucleus interrupt vector and some Nucleus code in order to identify the ready and sleeping tasks. If you somehow destroy the Nucleus interrupt vector or the required code, the System Debugger can't identify the ready and sleeping tasks.

The most common reasons for this type of error are:

- Pressing the RESET switch during debugging.
- Not initializing the Nucleus interrupt vector.
- Tasks writing over the Nucleus code.
- Tasks writing over iRMX 86 objects.

If any of these problems apply to your system, you must re-initialize your system.

EXAMPLE

If you want to display a list of all the ready and sleeping tasks in your system, enter:

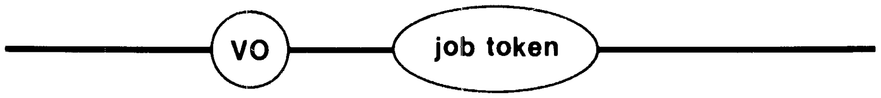
```
.VK
```

In this example, the System Debugger responds as follows:

```
Ready tasks:    4F02
Sleeping tasks: 56F5    558A    56BF    5204    51B3    5090    55EC    5052
                5021    4FFE    5697    5238    511F    566E    563A    5769
                50D1    2302
```

VO--Display Objects In A Job

The VO command displays the tokens for the objects in a job.



x-462

PARAMETER

job token	The token for the job whose objects you want to display.

DESCRIPTION

The VO command lists the tokens for a job's child jobs, tasks, mailboxes, semaphores, regions, segments, extensions, and composites in the format shown in Figure 4-6.

```
Child jobs:     xxxx   xxxx   xxxx  ...
Tasks:          xxxx   xxxx   xxxx  ...
Mailboxes:      xxxx   xxxx   xxxx  ...
Semaphores:     xxxx   xxxx   xxxx  ...
Regions:        xxxx   xxxx   xxxx  ...
Segments:       xxxx   xxxx   xxxx  ...
Extensions:     xxxx   xxxx   xxxx  ...
Composites:     xxxx   xxxx   xxxx  ...
```

Figure 4-6. Format Of VO Output

The fields in Figure 4-6 are as follows:

Child jobs	The tokens for the specified job's offspring jobs.
Tasks	The tokens for the tasks in the specified job.

Mailboxes	The tokens for the mailboxes within the job. A lower-case "o" immediately following a mailbox token means that one or more objects are queued at the mailbox. A lower-case "t" immediately following a mailbox token means that one or more tasks are queued at the mailbox.
Semaphores	The tokens for the semaphores in the specified job. A lower-case "t" immediately following a semaphore token means that one or more tasks are queued at the semaphore.
Regions	The tokens for the regions in the specified job. A lower-case "b" (busy) immediately following a region token means that a task has access to information guarded by the region.
Segments	The tokens for the segments in the specified job.
Extensions	The tokens for the extensions in the specified job.
Composites	The tokens for the composites in the specified job.

ERROR MESSAGES

The System Debugger returns the following error messages for the VO command

Error Message	Description
Syntax Error	You did not specify a parameter for the command or you made an error in entering the command.
TOKEN is not a Job	You entered a valid token that is not a job token.
*** INVALID TOKEN ***	The value you entered for the token is not a valid token.

EXAMPLE

Suppose you want to look at the objects in "51CE."

```
.VO 51CE
```

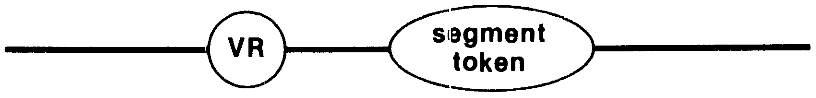
The System Debugger responds with the following display:

```
    Child jobs:     4F9F
    Tasks:          511F     50D1     5090     5052     5021     4FFE
    Mailboxes:      5119     5110     5100 t   50FB t   50CE t   5089 t
    Semaphores:     50FE     501F t
    Regions:
    Segments:       510C     5103     508C     504E     4FE6     4FCB
    Extensions:
    Composites:     511C     5113     50C8     5083     4FF3     4FED
```

The previous display shows the tokens for the child jobs, tasks, mailboxes, semaphores, regions, segments, extensions, and composites in the job. It also tells you that there are tasks waiting at four mailboxes and at one semaphore.

VR--Display I/O Request/Result Segment

The VR command displays information about the iRMX 86 Basic I/O System I/O request/result segment (IORS) that corresponds to the segment token that you enter.



x-463

PARAMETER

Segment token	The token for a segment containing the IORS you want to display. This segment must be an IORS or the VR command returns invalid information.

DESCRIPTION

The VR command displays the names and values for the fields of a specific IORS. The System Debugger cannot determine whether the segment contains a valid IORS, so it is up to you to ensure that the segment does indeed contain an IORS. If the parameter is a valid segment token for a segment containing an IORS, the System Debugger displays information about the IORS as shown in Figure 4-7.

The contents of the IORS pertain to the most recent I/O operation in which this IORS was used. For more information concerning the following fields, see the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O SYSTEMS.

```
I/O Request Result Segment

Status             xxxx         Unit status       xxxx
Device             xxxx         Unit              xx
Function           xxxxx        Subfunction       xxxxxxx
Count              xxxxxxx      Actual            xxxx
Device location    xxxxxxxx     Buffer pointer    xxxx:xxxx
Resp mailbox       xxxx         Aux pointer       xxxx:xxxx
Link forward       xxxx:xxxx    Link backward     xxxx:xxxx
Done               xxxx         Cancel ID         xxxx
Connection token   xxxx
```

Figure 4-7. Format Of VR Output

The fields in Figure 4-7 are as follows:

Status	The condition code for the I/O operation. See the description of I/O Request/Result Segments in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for further information.
Unit status	Additional status information. The contents of this field are meaningful only when the Status field is set to the E$IO condition (002BH). See the description of I/O Request/Result Segments in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for further information.
Device	The number of the device for which the last request was intended.
Unit	The number of the unit for which this request was intended.
Function	The operation that was performed by the Basic I/O System. The possible functions are as follows:

Function	System Call
Read	RQAREAD
Write	RQAWRITE
Seek	RQASEEK
Special	RQASPECIAL
Att Dev	RQ$A$PHYSICAL$ATTACH$DEVICE
Det Dev	RQ$A$PHYSICAL$DETACH$DEVICE
Open	RQAOPEN
Close	RQACLOSE

If the function field contains an invalid value, the System Debugger displays the value in this field, followed by a space and two question marks.

Subfunction	A further specification of the function that applies only when the Function field contains "Special." The possible subfunctions and their descriptions are as follows:

Subfunction (con't)	

Subfunction	Description
For/Que	Format or Query
Satisfy	Stream file satisfy function
Notify	Notify function
Device char	Device characteristics
Get Term Attr	Get terminal attributes
Set Term Attr	Set terminal attributes
Signal	Signal function
Rewind	Rewind tape
Read File Mark	Read file mark on tape
Write File Mark	Write file mark on tape
Retention Tape	Take up slack on tape

If the Function field doesn't contain "Special", then the Subfunction field contains "N/A." If the Subfunction field contains an invalid value, the System Debugger displays the value of the field followed by a space and two question marks.

Count	The number of bytes of data called for in the I/O request.
Actual	The number of bytes of data transferred in response to the request.
Device location	The eight-digit hexadecimal address of the byte where the I/O operation began on the specified device.
Buffer pointer	The address of the buffer from which the Basic I/O System read or to which it wrote in response to the request.
Resp mailbox	A token for the response mailbox to which the device sent the IORS after the operation.
Aux pointer	The pointer to the location of auxiliary data, if any. This field is significant only when the Function field contains "Special."
Link forward	The address of the next IORS in the queue where the IORS waited to be processed.
Link backward	The address of the previous IORS in the queue where the IORS waited to be processed.
Done	This field is always present but applies only to IORS's for I/O operations on random-access devices. When applicable, it indicates whether the I/O operation has been completed. The possible values are TRUE (FFFFH) and FALSE (0000H).

Cancel ID	A word that is used by device drivers to identify I/O requests that need to be cancelled. A value of 0 indicates a request that cannot be cancelled.
Connection token	The token for the file connection that was used to issue the request for the I/O operation.

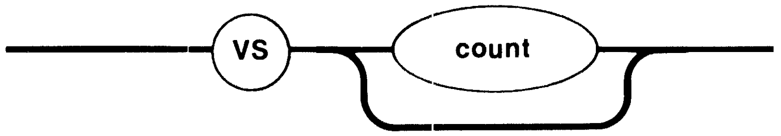
ERROR MESSAGES

The System Debugger returns the following error messages for the VR command:

Error Message	Description
Syntax Error	You did not specify a parameter for the command or you made an error in entering the command.
TOKEN is not a Segment	You entered a valid token that is not a segment token.
*** INVALID TOKEN ***	The value you entered for the token is not a valid token.
Segment wrong size, not an IORS	The specified segment is neither four nor five paragraphs in length, so it is not an I/O request/result segment.

VS--Display Stack And System Call Information

The VS command identifies system calls (as does the VC command) and displays the stack.



x-464

PARAMETER

count	A decimal or hexadecimal value that specifies the number of words from the stack that are to be included in the display. A suffix of T, as in 16T, means decimal. No suffix or a suffix of H indicates hexadecimal. If you do not specify a count, VS assumes the default value, 10H.

DESCRIPTION

The VS command identifies iRMX 86 system calls for all iRMX 86 subsystems (as does the VC command) and interprets the parameters on the stack. If a parameter is a string, the System Debugger displays the string. See the appropriate iRMX 86 manual for additional information about system calls.

The VS command interprets the CALL instruction at the current CS:IP. If you want to interpret a CALL instruction at a different CS:IP value, you must move the CS:IP to that value by using the iSBC 957B, iSDM 86, or iSDM 286 GO command.

The VS command uses current values of the SS:SP (stack segment:stack pointer) registers to display the current stack values. If the instruction is an iRMX 86 system call, VS displays the system call and the stack information, as shown in Figure 4-8.

```
xxxx:xxxx     xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx
xxxx:xxxx     xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx

S/W int:  xx  (subsystem)     entry code xxxx     system call

              :parameters.:
```

Figure 4-8. Format Of VS Output

The fields in Figure 4-8 are as follows:

xxxx:xxxx	The contents of the SS:SP.
xxxx	Stack values.
parameters	The names of the stack values. The parameters correspond to the stack values directly above them.

The three remaining fields in Figure 4-8 are identical to those in the VC command.

S/W int: xx (subsystem)	The software interrupt number and the iRMX 86 subsystem that corresponds to that number.
entry code xxxx	The entry code for the system call within the subsystem.
system call	The name of the iRMX 86 system call.

ERROR MESSAGES

The System Debugger always displays the words at the top of the stack. If it encounters problems, it then returns one of the following error messages.

Error Message	Description
Syntax Error	You made an error in entering the command.
Not a system CALL	The CS:IP is pointing to a CALL instruction that is not a system call.

Unknown entry code	This message indicates that one of two infrequent events has occurred. One is that the System Debugger has mistaken an operand for the software interrupt (INT) instruction. The other possibility is that a software link from user code into iRMX 86 code has been corrupted.

If the instruction is not a CALL instruction, VS displays the contents of the words on the stack and no message.

EXAMPLES

Suppose that by some means, such as the X command of the iSBC 957B, iSDM 86, or iSDM 286 monitor, you determine that the SS:SP is 4906:07CA. Suppose further that you then you use the VS command, as follows:

```
    .VS

   4906:07CA      0008   4984   4EAC   4983   4983   0000   0600   4906
   4906:07DA      49A4   0020   2581   4EAC   4EA1   4EE7   0000   0000

   S/W int: B8 (Nucleus)   entry code  0301   delete mailbox

                  :..excep$p..:.mbox.:
```

The parameter names identify the stack values directly above them. That is, the "excep$p" parameter name signifies that the first two words represent a pointer (4984:0008) to the exception code. Similarly, the "mbox" parameter signifies that the third word (4EAC) is the token for the mailbox being deleted.

Now, suppose that you move the SS:SP to 4906:07D0. If you invoke the VS command now, the debug monitor displays the stack as follows:

```
    .VS

    4906:07D0   4983   4983   0000   0600   4906   49A4   0020   2581
    4906:07E0   F7C7   F7C7   F5C7   F5C7   F5C7   F5C7   F5CF   F5CF

    Not a system CALL
```

The System Debugger displays the stack and a message which informs you that the instruction is a CALL instruction but is not a system call.

When an iRMX 86 system call is executed, its parameters are pushed onto the current stack, and then a CALL instruction is issued with the appropriate address. If you want to display the stack at such a call when there are more parameters than will fit on one line, the System Debugger automatically displays multiple lines of words from the stack, with corresponding lines of parameter description directly below them.

For example, suppose that you use the VS command as follows:

```
    .VS

   57CC:0F9A      015A   60C7   0000   60C6   60C6   0000   0600   57CC
   57CC:0FAA      60EF   0028   2322   0000   60C7   6618   6605   6623
   57CC:0FBA      6609   5A5F   5AF8   660B   0000   0000   0000   0000

   S/W Int:  B8 (Nucleus)  entry code 0100    create job

                  :...excep$p...:t$flgs:stksze:..sp..:..ss..:..ds..:..ip..:
                  :..cs..:.pri..:j$flgs:.exp$info$p..:maxpri:maxtsk:maxobj:
                  :poolmx:poolmn:param.:dirsiz:
```

This display indicates that the CALL instruction is a Nucleus RQ$CREATE$JOB system call having 18 parameters. The names of these parameters are shown between the colons (:). As usual, the words on the stack correspond to the parameters shown directly below those words.

The following display indicates that the CALL instruction is a Basic I/O System (BIOS) RQ$A$ATTACH$FILE system call having five parameters. The "subpath$p" parameter points to a string that is seven characters long. This string consists of the word "example."

```
    .VS

    57CC:0F4E     0F8C   57CC   65FD   0000   6600   69A2   0000   6602
    57CC:0F5E     660B   3C13   6602   2325   66D2   0F7C   0DF7   FFFF

    S/W Int:  C0 (BIOS)  entry code 0002     attach file

                  :...excep$p...:.mbox.:..subpath$p..:prefix:.user.:
  subpath-->07'example'
```

The following display indicates that the CALL instruction at CS:IP is an Extended I/O System RQ$S$RENAME$FILE system call having three parameters. There are two parameters with strings in this example. The new$path$p parameter points to a string that is four characters long. This string contains "XY70." The path$p parameter points to a string that is also four characters long and contains "temp."

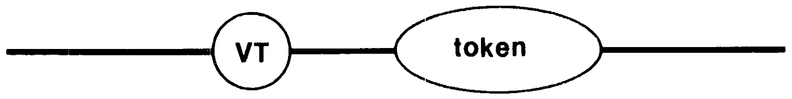
```
    .VS

   57CC:0F98      014A   60C7   06A5   60EF   06A5   60EF   0000   0600
   57CC:0FA8      57CC   60EF   0028   2322   0000   60C7   000A   6605

   S/W Int:  C1 (EIOS)  entry code 0108     rename file

                  :...excep$p...:.new$path$p..:...path$p....:
   new path-->04'XY70'
   path-->04'temp'
```

NOTE

If a string is more than 50 characters in length, the System Debugger will display only the first 50 characters of the string. And if the pointer to a string is 0000:0000, the System Debugger does not display the string.

VT--Display An iRMX 86 Object

The VT command displays information about the iRMX 86 object associated with the token you enter.



x-465

PARAMETER

token	The token for the object for which the System Debugger will display information.

DESCRIPTION

The VT command ascertains the type of object represented by the token and displays information about that object. Both the information and the format in which the System Debugger displays the information depend entirely upon the type of the object. The following sections are divided into display groups. Each display group illustrates the format of the display for a particular type of object.

ERROR MESSAGES

The System Debugger returns the following error messages for the VT command

Error Message	Description
Syntax Error	You did not specify a parameter for the command or you made an error in entering the command.
*** INVALID TOKEN ***	The value you entered for the token is not a valid token.

JOB DISPLAY

If the parameter that you supply is a valid job token, the System Debugger displays information about the job that has that token, as Figure 4-9 shows.

```
Object type = 1 Job

Current tasks    xxxx        Max tasks      xxxx    Max priority   xx
Current objects  xxxx        Max objects    xxxx    Parameter obj  xxxx
Directory size   xxxx        Entries used   xxxx    Job flags      xxxx
Except handler   xxxx:xxxx   Except mode    xx      Parent job     xxxx
Pool min         xxxx        Pool Max       xxxx    Initial size   xxxx
Pool size        xxxx        Allocated      xxxx    Largest seg    xxxx
```

Figure 4-9. Format Of VT Output (Job Display)

The fields in Figure 4-9 are as follows:

Current tasks	The number of tasks currently existing in the job.
Max tasks	The maximum number of tasks that can exist in the job at the same time. This value was set when the job was created with the RQ$CREATE$JOB system call.
Max priority	The maximum (numerically lowest) priority allowed for any one task in the job. This value was set when the job was created.
Current objects	The number of objects currently existing in the job.
Max objects	The maximum number of objects that can exist in the job at the same time. This value was set when the job was created.
Parameter obj	The token for the object that the parent job passed to this job. This value was set when the job was created.
Directory size	The maximum number of entries the job can have in its object directory. This value was set when the job was created.
Entries used	The number of objects currently cataloged in the job's object directory.
Job flags	The job flags parameter that was specified when the job was created.
Except handler	The start address of the job's exception handler. This address was set when the job was created.

Except mode	The value that indicates when control is to be passed to the new job's exception handler. This value was set when the job was created.
Parent job	The token for the job's parent.
Pool min	The minimum size (in 16-byte paragraphs) of the job's memory pool. This value was set when the job was created.
Pool max	The maximum size (in 16-byte paragraphs) of the job's memory pool. This value was set when the job was created.
Initial size	The initial size (in 16-byte paragraphs) of the job's memory pool.
Pool size	The current size (in 16-byte paragraphs) of the job's memory pool.
Allocated	The number of currently-allocated 16-byte paragraphs in the job's memory pool.
Largest Seg	The number of 16-byte paragraphs in the largest contiguous portion of the job's memory pool.

TASK DISPLAY

The System Debugger displays information about tasks in two different ways. The first display is for non-interrupt tasks and the second is for interrupt tasks. The format of the two types of tasks is shown in Figures 4-10 and 4-11.

```
object type = 2 Task

Static pri     xx        Dynamic pri    xx     Task state         xxxx
Suspend depth  xx        Delay req      xxxx   Last exchange      xxxx
Except handler xxxx:xxxx Except mode    xx     Task flags         xx
Containing job xxxx      Interrupt task no     Kernel saved ss:sp xxxx:xxxx
```

Figure 4-10. Format Of VT Output (Non-Interrupt Task)

```
object type = 2 Task

Static pri      xx        Dynamic pri     xx     Task state         xxxx
Suspend depth   xx        Delay req       xxxx   Last exchange      xxxx
Except handler  xxxx:xxxx Except mode     xx     Task flags         xx
Containing job  xxxx      Interrupt task  yes    Int Level          xx
Master mask     xx        Slave mask      xx     Slave number       xx
Pending int     xx        Max interrupts  xx     Kernel saved ss:sp xxxx:xxxx
```

Figure 4-11. Format Of VT Output (Interrupt Task)

The fields in Figures 4-10 and 4-11 are as follows:

Static pri	The current priority of the task. This value was set when the task's containing job was created.
Dynamic pri	A temporary priority that the Nucleus sometimes assigns to the task in order to improve system performance.
Task state	The state of the task. The five possible states, as they are displayed, are:

State	Description
ready	ready for execution
asleep	task is asleep
susp	task is suspended
aslp/susp	task is both asleep and suspended
deleted	task is being deleted

If this field contains an invalid value, the System Debugger displays the value followed by a space and two question marks.

Suspend depth	The number of RQ$SUSPEND$TASK system calls that have been applied to this task without corresponding RQ$RESUME$TASK system calls.
Delay req	The number of sleep units the task requested when it last specified a delay at a mailbox or semaphore, or when it last called RQ$SLEEP. If the task has not done any of these things, this field contains 0.
Last exchange	The token for the mailbox, region, or semaphore at which the task most recently began to wait.

COMMANDS

Except handler	The start address of the job's default exception handler. This value was set either when the task was created, by means of RQ$CREATE$TASK or RQ$CREATE$JOB, or later by means of RQ$SET$EXCEPTION$HANDLER.
Except mode	The value used to indicate the exceptional conditions under which control is to be passed to the new task's exception handler. This value was set either when the task was created, by means of RQ$CREATE$TASK or RQ$CREATE$JOB, or later by means of RQ$SET$EXCEPTION$HANDLER.
Task flags	The task flags parameter used when the task was created with the RQ$CREATE$TASK system call.
Containing job	The token of the job that contains this task.
Interrupt task	"No" signifies that the task is not an interrupt task. In this case, there are no fields following this field in the display. (See Figure 4-10.) "Yes" signifies that the task is an interrupt task. In this case, there are additional fields in the display. (See Figure 4-11.)
Kernel saved ss:sp	The contents of the ss:sp registers when the task last left the ready state.
Int level	The level that the interrupt task services. This level was set when this task called RQ$SET$INTERRUPT.
Master mask	The value associated with the interrupt mask for the master interrupt controller. This value represents the master interrupt levels that are disabled by virtue of the interrupt level that the task services. For example, if the task services interrupt level 51 (in octal), then master levels 6 and 7 are disabled, so the master mask field is 11000000B (=C0H). For more information concerning interrupt levels, see the iRMX 86 NUCLEUS REFERENCE MANUAL.

Slave mask	The value associated with the interrupt mask for a slave interrupt controller. This value represents the slave interrupt levels that are disabled by virtue of the level that the task services. For example, if the task services interrupt level 51 (octal), then slave levels 2 through 7 are disabled, so the slave level field is 11111100B (=FCH). For more information concerning interrupt levels, see the iRMX 86 NUCLEUS REFERENCE MANUAL.
Slave number	The programmable interrupt controller number of the slave that is referred to by the slave mask. This value depends entirely upon the interrupt level that the task services. If the value in the Int level field (after conversion to octal) is xy, then y+1 is the value in this field.
Pending int	The number of RQ$SIGNAL$INTERRUPT calls that are pending for this level.
Max interrupts	The maximum number of RQ$SIGNAL$INTERRUPT calls that can be pending for this level.

MAILBOX DISPLAY

The System Debugger displays information about mailboxes in three different ways. The first display appears when nothing is queued at the mailbox, the second appears when tasks are queued at the mailbox, and the third appears when objects are queued at the mailbox. The formats of the three types of display are shown in Figures 4-12, 4-13, and 4-14.

```
Object type = 3 Mailbox

Task queue head      xxxx              Object queue head     xxxx
Queue discipline     xxxx              Object cache depth    xxxx
Containing job       xxxx
```

Figure 4-12. Format Of VT Output (Mailbox With No Queue)

```
Object type = 3 Mailbox

Task queue head      xxxx                 Object queue head     xxxx
Queue discipline     xxxx                 Object cache depth    xxxx
Containing job       xxxx

Task queue         xxxx    xxxx  ...
```

Figure 4-13. Format Of VT Output (Mailbox With Task Queue)

```
Object type = 3 Mailbox

Task queue head      xxxx                 Object queue head     xxxx
Queue discipline     xxxx                 Object cache depth    xxxx
Containing job       xxxx

Object queue       xxxx    xxxx  ...
```

Figure 4-14. Format Of VT Output (Mailbox With Object Queue)

The fields in Figure 4-12, 4-13, and 4-14 are as follows:

Task queue head	The token for the task at the head of the queue. If the task queue for this mailbox is empty, this field contains 0.
Object queue head	The token for the object at the head of the queue. If the object queue for this mailbox is empty, this field contains 0.
Queue discipline	The manner in which tasks are queued at the mailbox. Tasks are queued "first-in/first-out" (FIFO) or by priority (PRI), depending upon how the mailbox was specified to RQ$CREATE$MAILBOX.
Object cache depth	The size of the high performance portion of the object queue that is associated with the mailbox. This size was specified when the mailbox was created with RQ$CREATE$MAILBOX.
Containing job	The token for the job that contains this mailbox.

Task queue	A list of tokens for the tasks queued at the mailbox in the order in which tasks are queued. If no tasks are queued at the mailbox, this list does not appear.
Object queue	A list of tokens for the objects queued at the mailbox in the order in which the objects are queued. If no objects are queued at the mailbox, this list does not appear.

SEMAPHORE DISPLAY

The System Debugger displays information about semaphores in two ways. The first display appears when no tasks are queued at the semaphore, and the second appears when tasks are queued at the semaphore. The formats for the two types of displays are shown in Figures 4-15 and 4-16.

```
Object type = 4 Semaphore

Task queue head     xxxx                Queue discipline    xxx
Current value       xxxx                Maximum value       xxxx
Containing job      xxxx
```

Figure 4-15. Format Of VT Output (Semaphore With No Queue)

```
Object type = 4 Semaphore

Task queue head     xxxx                Queue discipline    xxx
Current value       xxxx                Maximum value       xxxx
Containing job      xxxx

Task queue        xxxx    xxxx  ...
```

Figure 4-16. Format Of VT Output (Semaphore With Task Queue)

The fields in Figures 4-15 and 4-16 are as follows:

Task queue head	The token for the task at the head of the queue.

Queue discipline	The manner in which tasks are queued at the semaphore. The tasks are queued "first-in/first-out" (FIFO) or by priority (PRI), depending upon how the semaphore was specified when created with RQ$CREATE$SEMAPHORE.
Current value	The number of units currently held by the semaphore.
Maximum value	The maximum number of units the semaphore can hold. This number was specified when the semaphore was created with RQ$CREATE$SEMAPHORE.
Containing job	The token for the job that contains the semaphore.
Task queue	A list of tokens for the tasks queued at the semaphore, in the order in which the tasks are queued there. If no tasks are queued at the semaphore, this list does not appear.

REGION DISPLAY

If the parameter that you supply is a valid token for a region, the System Debugger displays information about the associated region as shown in Figure 4-26.

```
Object type = 5 Region

Entered task       xxxx                    Queue discipline     xxxx
Containing job     xxxx

Task queue       xxxx     xxxx  ...
```

Figure 4-17. Format Of VT Output (Region)

The fields in Figure 4-17 are as follows:

Entered task	The token for the task that is currently accessing information guarded by the region.
Queue discipline	The manner in which tasks are queued at the region. The tasks are queued first-in/first-out (FIFO) or by priority (PRI), depending upon how the region was specified when created with RQ$CREATE$REGION.

Containing job	The token for the job that contains the region.
Task queue	Tokens for the tasks waiting to gain access to data guarded by the region. This line is displayed only if a task already has access to the data guarded by the region.

SEGMENT DISPLAY

If the parameter that you supply is a valid token for a segment, the System Debugger displays information about the associated segment as shown in Figure 4-18.

```
Object type = 6 segment

Num of paragraphs     xxxx                Containing job      xxxx
```

Figure 4-18. Format Of VT Output (Segment)

The fields in Figure 4-18 are as follows:

Num of paragraphs	The number of 16-byte paragraphs in this segment. The size of the segment was specified when the segment was created with the RQ$CREATE$SEGMENT system call.
Containing job	The token for the job that contains the segment.

EXTENSION OBJECT DISPLAY

If the parameter that you supply is a valid token for an extension, the System Debugger displays information about the associated extension as shown in Figure 4-19.

```
Object type = 7 Extension

Extension type     xxxx                   Deletion mailbox     xxxxx
Containing job     xxxx
```

Figure 4-19. Format Of VT Output (Extension Object)

The fields in Figure 4-19 are as follows:

Extension type	The type code associated with composite objects licensed by this extension. This code was specified when the RQ$CREATE$EXTENSION system call, was used to create this extension type. See the iRMX 86 NUCLEUS REFERENCE MANUAL for more information concerning extension types.
Deletion mailbox	The token for the deletion mailbox associated with this extension. This mailbox was specified when the RQ$CREATE$EXTENSION system call was used to create this extension type.
Containing job	The token for the job that contains the extension.

COMPOSITE OBJECT DISPLAY

There are five kinds of composite displays. The first kind depicts all composites except those defined in the Basic I/O System (BIOS). The second kind depicts BIOS user objects. The remaining kinds depict BIOS physical, stream, and named file connections.

The format for the display of non-BIOS objects is as shown in Figure 4-20.

```
Object type = 8 Composite

Extension type  xxxx    Extension obj    xxxx    Deletion mbox    xxxx
Containing job  xxxx    Num of entries   xxxx

Component list  xxxx    xxxx    xxxx    xxxx  ...
```

Figure 4-20. Format Of VT Output (Composite Object Other Than BIOS)

The fields in Figure 4-20 are as follows:

Extension type	The code for the extension type of the extension object used to create this composite. This code was specified when the extension object was created with RQ$CREATE$EXTENSION.
Extension obj	The token for the extension object used to create this composite object.

Deletion mbox	The token for the mailbox to which this composite goes when the composite is to be deleted. This mailbox was specified when the extension was created with RQ$CREATE$EXTENSION.
Containing job	The token for the job that contains the composite object.
Num of entries	The number of component entries in the composite object.
Component list	The list of tokens for the components of the composite.

The format for the Basic I/O System user object display is shown in Figure 4-21.

```
Object type = 8 Composite

Extension type   xxxx    Extension obj  xxxx   Deletion mbox    xxxx
Containing job   xxxx    Num of entries xxxx

   BIOS USER OBJECT:
User segment     xxxx    Number of IDs  xxxx

User ID list     xxxx    xxxx  ...
```

Figure 4-21. Format Of VT Output (BIOS User Object Composite)

The new fields introduced in Figure 4-21 are as follows:

User segment	The token for the segment containing the user IDs for the user object.
Number of IDs	The number of user IDs associated with the user object.
User ID list	List of the user IDs associated with the user object.

The format for a (file) connection to a physical file is shown in Figure 4-22.

```
Object type = 8 Composite

Extension type   xxxx Extension obj    xxxx     Deletion mbox  xxxx
Containing job   xxxx Num of entries   xxxx

      T$CONNECTION OBJECT
File driver     Physical Conn flags       xx          Access         xxxx
Open mode       xxxx     Open share       xxxx        File pointer   xxxxxxxx
IORS cache      xxxx     File node        xxxx        Device desc    xxxx
Dynamic DUIB    xxxxx    DUIB pointer     xxxx:xxxx Num of conn      xxxx
Num of readers  xxxx     Num of writers   xxxx        File share     xxxxxxx
File drivers    xxxx     Device gran      xxxx        Device size    xxxxxxxx
Device functs   xxxx     Num dev conn     xxxx        Device name    xxxxxxxxxx
```

Figure 4-22. Format Of VT Output (Physical File Connection)

The new fields introduced in Figure 4-22 are as follows:

File driver	The BIOS file driver to which this connection is attached. The three possible values are physical, stream, and named. If this field contains an invalid value, the System Debugger displays the value followed by a space and two question marks.
Conn flags	The flags for the connection. If bit 1 is set to one, this connection is active and can be opened. If bit 2 is set to one, this is a device connection. (Bit 0 is the low-order bit.)
Access	The access rights for this connection. This display uses a single character to represent a particular access right. If the file has the access right, the character appears. However, if the file does not have the access right, a hyphen (-) appears in the character position. The access rights, along with the characters that represent them, are as follows:

```
                   |------- Delete
                   ||------ List
Directory files:   |||----- Add
                   ||||---- Change
                   DLAC

                   DRAU
                   ||||---- Update
Data Files:        |||----- Append
                   ||------ Read
                   |------- Delete
```

Open mode	The mode established when this connection was opened. The possible values are:

Open Mode	Description
Closed	Connection is closed
Read	Connection is open for reading
Write	Connection is open for writing
R/W	Connection is open for reading and writing

If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks. If this value is Read, Write, or R/W, this value was specified when the connection was opened.

Open share	The sharing status established for this connection when it was opened. The sharing status for a connection is a subset of the sharing status of the file (see the File share field). The possible values are:

Share Mode	Description
Private	Private use only
Readers	File can be shared with readers
Writers	File can be shared with writers
ALL	File can be shared with all users

If the connection is not open, then 0 is displayed. If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks. This probably indicates that the connection data structure has been corrupted.

File pointer	The current location of the file pointer for this connection.
IORS cache	The token for the segment at the head of the BIOS list of used IORS's. These IORS's are being saved for the RQ$WAIT$IO system call to use again. The list is empty if 0000 appears in this field.
File node	The token for a segment that the Operating System uses to maintain information about the connection. The information in this segment appears in the next two fields.
Device desc	The token for the segment that contains the device descriptor. The device descriptor is used by the Operating System to maintain information about the connections to the device.

Dynamic DUIB	Indicates whether a DUIB was created dynamically when the device associated with this connection was attached.
DUIB pointer	The address of the Device Unit Information Block (DUIB) for the device unit containing the file. See the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O OPERATING SYSTEMS for more information about the DUIB.
Num of conn	The number of connections to the file.
Num of readers	The number of connections to the file that are currently open for reading.
Num of writers	The number of connections to the file that are currently open for writing.

File share

The share mode of the file. This parameter defines how other connections to the file can be opened. The share mode of a file is a superset of the sharing status of each of the connections to the file (see the Open share field). The possible values are:

Share Mode	Description
Private	Private use only
Readers	File can be shared with readers
Writers	File can be shared with writers
ALL	File can be shared with all users

If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks. This probably means that the internal data structure for the file or the fnode for the file has been corrupted. See the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about sharing modes for files and connections.

File drivers

The file drivers that the file can be connected to. If the file can be connected to a file driver, then the bit in the display is set to 1. Bit 0 is the rightmost bit.

Bit	Driver
0	Physical file
1	Stream file
2	reserved
3	Named file

Device gran	The granularity (in bytes) of the device. This is the minimum number of bytes that can be written to or read from the device in a single (physical) I/O operation.
Device size	The capacity (in bytes) of the device.
Device functs	Describes the functions supported by the device on which this file resides. Each bit in the low-order byte of the field corresponds to one of the possible device functions. If that bit is set to 1, then the corresponding function is supported by the device.

Bit	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

Num dev conn	The number of connections to the device.
Device name	The 14- (or fewer) character name of the device where this file resides.

The format for a (file) connection to a stream file is shown in Figure 4-23.

```
  Object type = 8 Composite

  Extension type  xxxx     Extension obj   xxxx       Deletion mbx   xxxx
  Containing job  xxxx     Num of entries  xxxx

       T$CONNECTION OBJECT
File driver     Stream   Conn.flags      xx         Access        xxxx
Open mode       xxxx     Open share      xxxx       File pointer  xxxxxxxx
IORS cache      xxxx     File node       xxxx       Device desc   xxxx
Dynamic DUIB    xxxxx    DUIB pointer    xxxx:xxxx  Num of conn   xxxx
Num of readers  xxxx     Num of writers  xxxx       File share    xxxxxxx
File drivers    xxxx     Device gran     xxxx       Device size   xxxxxxxx
Device functs   xxxx     Num dev conn    xxxx       Device name   xxxxxxxxxx
Req queued      xxxx     Queued conn     xxxx       Open conn     xxxx
```

Figure 4-23. Format Of VT Output (Stream File Connections)

The new fields introduced in Figure 4-23 are as follows:

Req queued	The number of requests that are currently queued at the stream file.
Queued conn	The number of connections that are currently queued at the stream file.
Open conn	The number of connections to the stream file that are currently open.

The format for a named (file) connection display is shown in Figure 4-24.

```
Object type = 8 Composite

Extension type    xxxx   Extension obj    xxxx   Deletion mbx     xxxx
Containing job    xxxx   Num of entries   xxxx

        T$CONNECTION OBJECT
File driver     Named    Conn flags       xx          Access           xxxx
Open mode       xxxx     Open share       xxxx        File pointer     xxxxxxxx
IORS cache      xxxx     File node        xxxx        Device desc      xxxx
Dynamic DUIB    xxxxx    DUIB pointer     xxxx:xxxx   Num of conn      xxxx
Num of readers  xxxx     Num of writers   xxxx        File share       xxxx
File drivers    xxxx     Device gran      xxxx        Device size      xxxxxxxx
Device functs   xxxx     Num dev conn     xxxx        Device name      xxxx
Num of buffers  xxxx     Fixed update     xxxx        Update timeout   xxxx
Fnode number    xxxx     File type        xxxx        Fnode flags      xxxx
Owner           xxxxx    File/Vol gran    xxxx        Fnode PTR(s)     xxxx:xxxx
Total blocks    xxxxxxxx Total size       xxxxxxxx    This size        xxxxxxxx
Volume gran     xxxx     Volume size      xxxxxxxx    Volume name      xxxxxx
```

Figure 4-24. Format Of VT Output (Named File Connection)

The new fields introduced in Figure 4-24 are as follows:

Num of buffers	The number of buffers allocated for blocking and unblocking I/O requests involving the device. A value of 0 indicates that the device is not a random-access device.
Fixed update	Indicates whether the device uses the fixed update feature. For more information about fixed updating, see the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

Update timeout	The length of the timeout for the update timeout feature, measured in Nucleus time units. For more information about update timeout, see the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.
Fnode number	The fnode number of this file. For more information about fnodes, see the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL.
File type	The type of named file. The possible values are:

File type	Description
DIR	Directory file
DATA	Data file
SPACEMAP	Volume free space map file
FNODEMAP	Free fnodes map file
BADBLOCKMAP	Bad blocks file

Fnode flags	A word containing flag bits. If a bit is set to 1, the following description applies. Otherwise, the description does not apply. (Bit 0 is the low-order bit.)

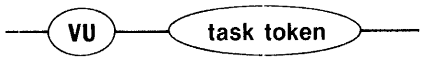
Bit	Description
0	This fnode is allocated
1	The file is a long file
2	Primary fnode
3-4	Reserved
5	This file has been modified
6	This file is marked for deletion
7-15	reserved

Owner	The ID of the owner of the file. If this field has a value of FFFF, then the field is interpreted as "World." See the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about owners of files.
File/Vol gran	The granularity of the file (in volume granularity units).
Fnode PTR(s)	The values of the fnode pointers. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information about fnode pointers.
Total blocks	The total number of volume blocks currently used for the file; this includes indirect blocks. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information about blocks.

Total size	The total size (in bytes) of the file; this includes actual data only. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
This size	The total number of bytes allocated to the file for data. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
Volume gran	The granularity (in bytes) of the volume.
Volume size	The size (in bytes) of the volume.
Volume name	The name of the volume.

VU--Display The System Calls In A Task's Stack

The VU command displays (unwinds) the iRMX 86 system calls in the stack of the task whose token you enter.



x 647

PARAMETER

token	The token for the task whose stack is to be searched for system calls.

DESCRIPTION

The VU command accepts a token for a task and then searches the task's stack for iRMX 86 system calls, starting at the top of the stack. For each system call it finds there, it displays two things:

- The address of the next instruction to be executed on behalf of the task after the system call has finished running. This is the return address for the call.
- The VS display with two lines of stack values (or more if required for parameters), as if the CALL instruction for the system call were in the CS:IP register and the displayed stack values were at the top of the stack.

This command requires that the task stack reside inside an iRMX 86 segment.

The VU command uses internal iRMX 86 data structures to get some of its information. Immediately after the system call at the top of the task's stack runs to completion, the data structures are updated. Therefore, there is a brief moment when the information the VU command uses is obsolete. This means that it is possible, although unlikely, that the first system call the VU command displays is not valid.

Figure 4-25 illustrates the format of one system call display by the VU command. System calls can be nested, with one calling another, so some invocations of the VU command produce multiple displays of the type shown in the figure. The example that follows illustrates this.

If there are no system calls in the stack of the indicated task, the VU command displays the message:

```
No system calls on stack
```

```
Return cs:ip - yyyy:yyyy
xxxx:xxxx       xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx
xxxx:xxxx       xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx

S/W int:  xx  (subsystem)     entry code xxxx     system call

                :parameters.:
```

Figure 4-25. Format Of VU Output

The fields in Figure 4-25 are as follows:

Return cs:ip	The return address for the system call of this display.
xxxx:xxxx	The address of the (top of the) portion of the stack that is devoted to this call.
xxxx	Stack values.
parameters	The parameter names associated with the stack values. The parameters correspond to the stack values directly above them.
S/W int: xx (subsystem)	The software interrupt number for this call and the iRMX 86 subsystem corresponding to that number.
entry code xxxx	The entry code for the system call within the subsystem.
system call	The name of the iRMX 86 system call.

ERROR MESSAGES

The System Debugger returns the following error messages for the VU command.

Error Message	Description
Syntax Error	You made an error in entering the command.
*** INVALID TASK TOKEN ***	The value you entered for the token is not a valid task token.
Stack not in an iRMX 86 segment	The stack of the indicated task is not in an iRMX 86 segment, as is required.

EXAMPLE

This example shows how the VU command responds when system calls are nested. The task for the example has called RQ$S$WRITE$MOVE of the Extended I/O System. RQ$S$WRITE$MOVE has called RQ$A$WRITE of the Basic I/O System. And RQAWRITE has called RQ$RECEIVE$MESSAGE to wait for the data transfer to be completed.

Before the message arrives signalling the completion of the transfer, the VU command is invoked, as follows:

```
.VU ????
```

The System Debugger responds by displaying the following:

```
Return cs:ip - 0104:576A
416A:01B2     01C8   416A   01CC   416A   FFFF   376E   8763   2988
416A:01C2     1550   0000   719E   2FF9   3440   E55E   5000   DD54

S/W Int: B8 (Nucleus)  Entry code  0303    Receive message

                :...excep$p...:....resp$p...:.time.:.mbox.:

Return cs:ip - 1756:08E7
416A:01D4     01EA   416A   3F56   0400   0000   42E9   429A   7866
416A:01E4     1430   5246   01FE   22F9   1400   021D   0000   01FE

S/W Int: C0 (BIOS)  Entry code  000E    Write

                :...excep$p...:.mbox.:.count:..buffer$p...:.conn.:

Return cs:ip - 3A98:06FA
416A:0218     0020   39F4   0400   0034   39F4   429A   5A84   9344
416A:0228     4456   0000   0000   C7C7   C7C7   C7C7   C7C7   C7C7

S/W Int: C1 (EIOS)  Entry code  0101    Write move

                :...excep$p...:.count:..buffer$p...:.conn.:
```

***

INDEX

iRMX™ 86 CRASH ANALYZER REFERENCE MANUAL

CONTENTS

CONTENTS (continued)

***

CHAPTER 1 INTRODUCTION

ORGANIZATION OF THE MANUAL

This manual is divided into four chapters. Some of the chapters contain introductory or overview material which you might not need to read if you are already familiar with the Crash Analyzer. Other chapters contain invocation information and reference material to which you can refer as you analyze the problems in your software following the failure of a system or an application program. You can use this section to determine which of the other chapters you should read.

The organization of the manual is as follows:

Chapter 1	This chapter describes the organization of the manual and introduces the Crash Analyzer. It describes the features and the environment of the Crash Analyzer. You should read this chapter if you are going through the manual for the first time.
Chapter 2	This chapter explains how to configure, initialize, and load the Dumper portion of the Crash Analyzer. You should read this chapter if you are going through the manual for the first time or if you need to re-load the Dumper.
Chapter 3	This chapter describes how to invoke the Crash Analyzer. You should read this chapter to learn how to invoke the Dumper and the Analyzer. You may also want to use this chapter as a reference to the options available when you invoke the Dumper or Analyzer.
Chapter 4	This chapter describes the formats and explains the fields in the print out that the Crash Analyzer generates. The individual formats are arranged in the order they appear in the print out. You should refer to this chapter during system analysis for specific information about the displays.

REASONS FOR USING THE CRASH ANALYZER

The Crash Analyzer aids you in debugging iRMX 86 applications. It provides you with an analysis of software problems following the failure either of your system or an application program in an iRMX 86 environment. The Crash Analyzer helps you to determine the reasons for system failure by:

- Producing a dump file containing a memory image of the crash situation.
- Analyzing the dump file and producing a detailed, formatted report of the crash situation.
- Listing system objects in detail and checking for inconsistencies where possible.

The following section further describes the parts of the Crash Analyzer and the environments in which they run.

PARTS OF THE CRASH ANALYZER

The Crash Analyzer is a single product consisting of two parts:

- The Dumper which produces a disk copy of a memory image. This copy is called the dump file. The Dumper runs in the iRMX 86 application system that you are debugging.
- The Analyzer which reads the dump file and creates a formatted printout file. This printout file contains clearly labeled, formatted information about system data structures. The Analyzer runs on a Series III Microcomputer Development System. It requires a secondary storage device to contain the dump file and the formatted report.

REQUIREMENTS OF THE CRASH ANALYZER

In order to use the Crash Analyzer, the iSDM 86 Monitor must be part of your system.

HOW THE CRASH ANALYZER IS SUPPLIED

The Crash Analyzer is available on Release Diskettes in ISIS-II format or iRMX 86 format. You must perform the loading and configuration of the Dumper on a Series III Microcomputer Development System. Therefore, if you use the iRMX 86 format, you may have to copy some files to an ISIS-II format.

***

CHAPTER 2 CONFIGURING, INITIALIZING, AND LOADING THE DUMPER

In order to use the Crash Analyzer, you must configure, initialize, and load the Dumper. This chapter describes the options available when you use the iRMX 86 Interactive Configurator (ICU) to configure the Dumper into your system. It then describes how to initialize and load the Dumper module from a microcomputer development system, using the iSDM 86 Monitor. The remainder of the chapter describes what to do if you must re-load the Dumper.

USING THE INTERACTIVE CONFIGURATOR TO CONFIGURE THE DUMPER

The iRMX 86 CONFIGURATION GUIDE explains how to use the ICU to include the Dumper in your system. One of the options you have when configuring you system is whether to locate the Dumper in RAM or ROM. After you configure the Dumper, be sure to look up the address for RQ$DUMP$BOOT$INIT in the Dumper locate map SDUMPR.MP2. You will need this address to re-initialize the dumper if you have to reset the system or if you accidentally destroy data structures necessary to the Dumper and the iSDM 86 Monitor.

INITIALIZING AND LOADING THE DUMPER

When you configure and bootstrap load the system, it automatically initializes and loads the Dumper. You are now prepared to run the Crash Analyzer if it is necessary. If a system program or an application should fail, all you have to do is activate the iSDM 86 Monitor on your screen and invoke the Dumper. A common way to bring up the iSDM 86 Monitor is to press the non-maskable interrupt. This procedure causes the iSDM 86 to display a prompt (.). When you invoke the Dumper, the system automatically initializes and loads the Dumper. See Chapter 3 to learn how to invoke the Dumper.

NOTE

Avoid using the reset switch. If you do use the reset switch, you will have to re-initialize and possibly reload the Dumper.

RE-INITIALIZING AND RE-LOADING THE DUMPER

If you have to reset the system, or if you accidentally destroyed some data structures necessary to the Dumper and the iSDM 86 Monitor, you can still use the Dumper to create a valid dump file. To do this, you must re-initialize and possibly reload the Dumper. Rather than re-loading your system, you can initialize the Dumper by using the iSDM 86 Monitor go command (G) and the address for RQ$DUMP$BOOT$INIT (as listed in the Dumper locate map SDUMPR.MP2).

```
    .G <bbbb>:<oooo>
```

The system responds with the Dumper sign-on message, a breakpoint, and a prompt as follows:

```
    iRMX 86 Dumper initialized

    *BREAK* at <bbbb>:<oooo>
```

where <bbbb> and <oooo> are base and offset addresses for internal iSDM 86 Monitor structures.

If the sign-on message does not appear, the system responds with the following error message:

```
    Bad Command
```

If the system returns a "Bad Command" error message, you must re-load the Dumper into memory by entering the iSDM 86 Monitor load (L) command at your microcomputer development system as follows:

```
    .L :fx:filename
```

where ":fx:" is the disk identifier that corresponds to the disk on which the ICU placed the Dumper and "filename" is the name of the file that contains the Dumper on the diskette.

After you have re-loaded the Dumper, you can enter the go command (as shown previously in this section) to re-initialize the Dumper.

***

CHAPTER 3
INVOKING THE CRASH ANALYZER

After you configure, initialize, and load the Dumper, you can invoke both the Dumper and the Analyzer portions of the Crash Analyzer any time you need them. (Refer to Chapter 2 for more information on configuring, initializing and loading.) This chapter presents a situation in which you would want to use the Dumper and the Analyzer. This situation is a general scenario of the steps you might take when an application fails. Following the scenario are detailed descriptions of how to invoke the Dumper and the Analyzer.

SCENARIO OF HOW TO USE THE CRASH ANALYZER

This section presents a general scenario of how, and in what kind of a situation, to use the Crash Analyzer. Your iRMX 86 System may differ slightly from the example used in Figure 3-1 but the procedure for using the Crash Analyzer is the same.

THE EQUIPMENT AND THE PROBLEM

Figure 3-1 shows a system consisting of the following equipment:

- A target system with an iAPX 86, 88-based processor board, memory, any necessary controllers, and a compatible terminal.
- A Series III Microcomputer Development System and a compatible printer.

Your system can be any iRMX 86 System but you must connect the Series III Microcomputer Development System to the target system with the iSDM 86 Monitor.

Suppose you are running an application on an iRMX 86 System and for some reason your application fails. You can use the Crash Analyzer to help find out why the application failed.

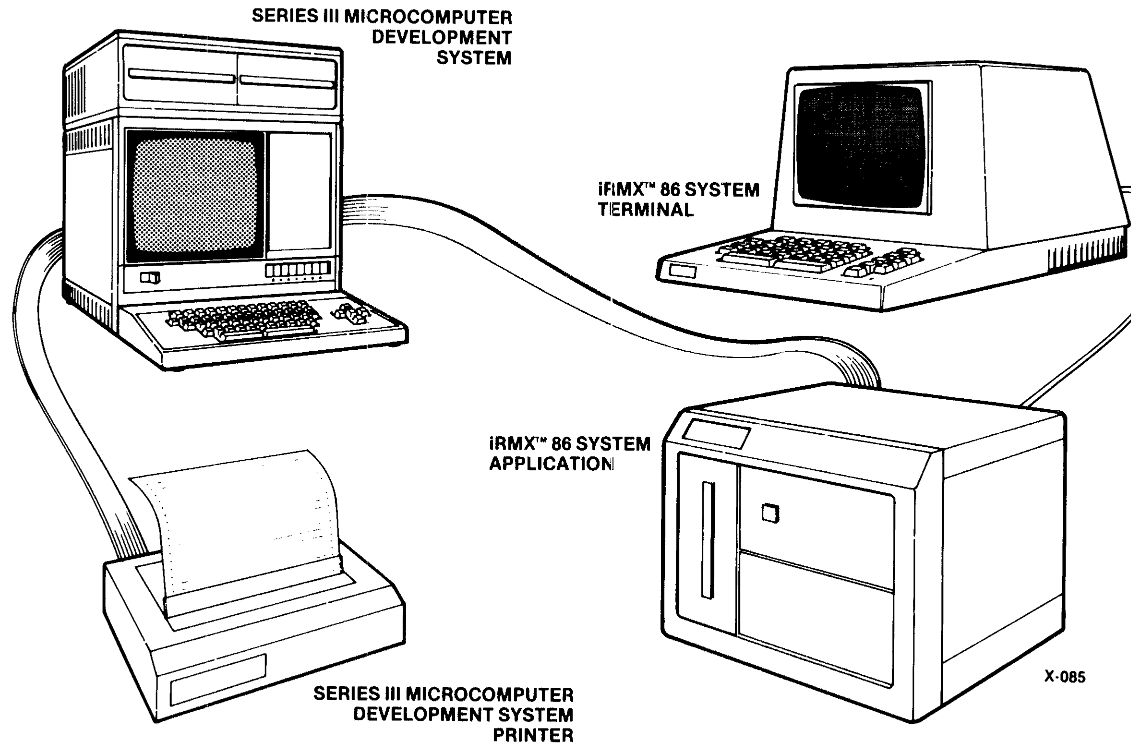

SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM
iRMX™ 86 SYSTEM TERMINAL
iRMX™ 86 SYSTEM APPLICATION
SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM PRINTER
X-085

Figure 3-1. Example System

USING THE CRASH ANALYZER

This section describes the general steps you should take when you want to use the Crash Analyzer. The steps refer you to detailed explanations of the specific invocations.

1. Activate the iSDM 86 Monitor and invoke the Dumper. A common way to bring up the iSDM 86 Monitor is to press the non-maskable interrupt on your target system.

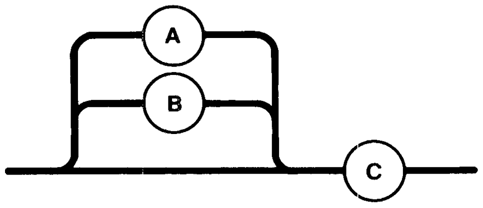
2. Invoke the Dumper on the target System. See "Invoking the Dumper" in this chapter. The Dumper uses the iSDM 86 link to create a disk file on the Series III Microcomputer Development System. This disk file (called the dump file) contains a copy of the system's memory.

3. Invoke the Analyzer on the Series III Microcomputer Development System. See "Invoking the Analyzer" in this chapter. The Analyzer reads the dump file and produces a formatted print file which it sends to the printer or a disk file. This print file contains clearly labeled information about the system data structures.

4. Use listings in Chapter 4 to help you understand the information in the print file.

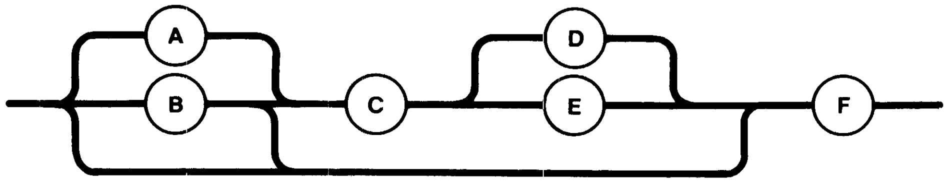
PICTORIAL REPRESENTATION OF SYNTAX

This manual uses a schematic device to illustrate the syntax of commands. The schematic consists of what looks like an aerial view of a model railroad setup, with syntactic entities scattered along the track. Imagine that a train enters the system at the left, drives around as much as it can or wants to (sharp turns and backing up are not allowed), and finally departs at the right. The command it generates in so doing consists, in order, of the syntactic entities that it encounters on its journey. The following pictorial syntax shows two ways (A or B) of reaching "C.":



x-116

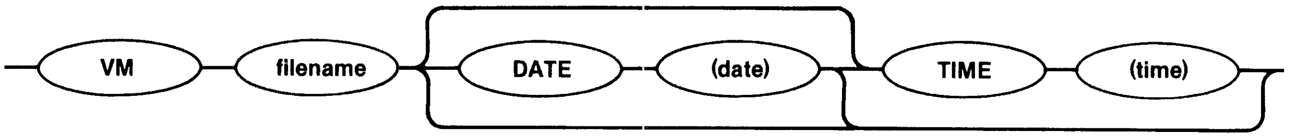
The schematics do get more complicated, but just remember that you can begin at any point on the left side of the track and take any route to get to the end as long as you do not back up. Some of the possible combinations of syntactic elements are: ACDF, BCEF, BF, AF, and F.



x-117

INVOKING THE DUMPER

You can invoke the dumper by interrupting into the iSDM 86 Monitor (on an iRMX 86 application system) and using the VM command.



X-086

PARAMETERS

filename	The name of the ISIS-II file to which you want to dump the disk copy of the system memory. The beginning portion of this name can consist of a logical name enclosed in colons (such as :F1:). This indicates the drive on which to place the file. If you omit the logical name, the Dumper places the file resides in the default drive (:F0:).
DATE	If you want the analysis header (explained in Chapter 4) to include a date, you must enter the word "DATE" immediately preceding the actual date.
(date)	The date that you invoke the Dumper. This parameter can be up to 20 characters in length and in any form you wish. The characters you enter for the date must be enclosed by parentheses. The date you enter is placed in the dumpfile and printed during analysis. The date is an optional parameter; if you do not specify a date, the Crash Analyzer omits the it in the analysis header. See Chapter 4 for more information about the analysis header.
TIME	If you want the Analysis Header (explained in Chapter 4) to include a time, you must enter the word "TIME" immediately preceding the actual time.
(time)	The time that you invoke the Dumper. This parameter can be up to 10 characters in length and in any form you wish. The characters you enter for the time must be enclosed by parentheses. The time that you enter is placed in the dumpfile and printed during analysis. The time is an optional parameter; if you do not specify a time, the Crash Analyzer omits the time in the analysis header. See Chapter 4 for more information about the analysis header.

The dumper displays the following message immediately after you invoke it:

```
Start iRMX 86 system dump V<x.x>
```

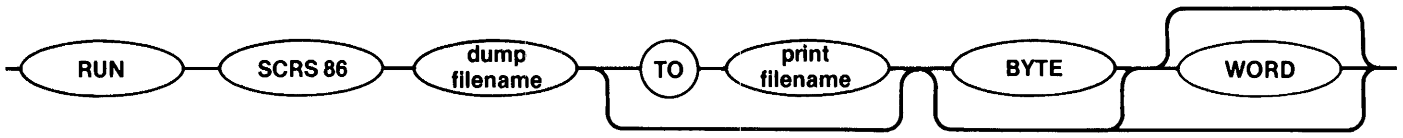
When the Dumper finishes creating the dump file, it displays the following message:

```
Dump complete to file <filename>
```

where <filename> is the file name you specified in the VM command. The iSDM 86 Monitor then issues a new prompt (.).

INVOKING THE ANALYZER

You can invoke the Analyzer on the Series III Microcomputer Development System by using the following command.



X-087

PARAMETERS

RUN	The Series III RUN command.
SCRS86	The name of the Analyzer.
dump-filename	The name of the file that is the source of the system memory image to be analyzed. This is the same file you specified when you invoked the Dumper.
TO	If you want to include a print file name, you must enter the word "TO" preceding the print file name you select.
print-filename	The name under which the Analyzer places the analyzed output. If you do not specify a "print-filename", the Analyzer uses the "dump filename" with "PRT" as the extension. Do not use the name of a device alone, unless the name specifies a device printer such as :LP:.

BYTE	An optional format in which the Analyzer may print the contents of the iRMX 86 segments. If you specify the BYTE option, the Analyzer prints the contents of the iRMX 86 segments in BYTE format. An example of the BYTE format is as follows:

```
contents:  BBBB:0000  xx  xx  xx...*aaaaaaaaaaaa*
```

where:

BBBB:0000	The base and offset address of the iRMX 86 segments.
xx	A pair of hexadecimal digits representing a byte.
a...a	The ASCII representation of the corresponding byte (if printable). If the byte value cannot be printed, the Analyzer places a period (.) in its place.

WORD	An optional format in which the Analyzer may print the contents of the iRMX 86 segments. If you specify the WORD option, the Analyzer prints the contents the iRMX 86 segments in hexadecimal WORD format. An example of the WORD format is as follows:

```
contents:  BBBB:0000  xxxx  xxxx  xxxx  xxxx...
```

where:

BBBB:0000	The base and offset address of the iRMX 86 segments.
xxxx	Four hexadecimal digits representing a word.

If you specify both BYTE and WORD, the iRMX 86 segments are displayed in both formats. If you do not specify either BYTE or WORD, the contents of the iRMX 86 segments do not appear in the print file.

ERROR MESSAGES

The following error messages appear on the your screen when you make an error in invoking the Analyzer or during a file operation. These errors cause the Analyzer to terminate all operations and display the error message.

Message	Description
Argument size exceeds 80 characters	When you invoked the Analyzer, one of the arguments exceeded 80 characters in length.
<filename>, error during <operation type> <filename>, EXCEPTION <nnnn>H <message>	The Analyzer encountered an exceptional condition when it tried to perform an operation on the file name. The <operation type> is one of the following: open create close detach read write seek The Analyzer also displays the exception code <nnnn>H and the mnemonic for the exception in <message>. Refer to the iRMX 86 Operator's Manual to find out what the exception codes mean.
<filename>, illegal file name	The file name you specified when you invoked the Analyzer is not a valid ISIS II file name.
<filename>, is not an iRMX 86 dump file	The file name you specified when you invoked the Analyzer refers to a file that was not originally created by the Dumper.
<filename>, no such file	The Analyzer cannot find the file you specified.
<keyword>, invalid keyword	You specified a format option other than WORD or BYTE when you invoked the Analyzer.
Non-blank delimiter in input string	When you invoked the Analyzer, you used a delimiter other than a blank. The only delimiter the Analyzer accepts is a blank.

Null dump file name	You did not specify a name for the dump file when you invoked the Analyzer.
Null output file name	When you invoked the Analyzer, you included "TO" but you did not specify a print file name.

***

CHAPTER 4
LISTING FORMAT

This chapter describes the format and explains the fields in the listing that the Analyzer outputs. These individual sections of the listing are arranged in the order they appear in the printout. For quick reference, this chapter includes a Table of Contents that lists the pages on which the different sections of the listing appear.

Note: this manual will not explain some of the outputs on the display field since those outputs are meant for Intel in-house use only.

LISTINGS

ANALYSIS HEADER

The section of the listing shown in Figure 4-1 identifies the file being dumped along with the time and date of the dump.

```
******************************************************************************
*
*   iRMX 86 Crash Analyzer   -   V<x.x>
*
*   Date:   <date of dump>
*
*   Time:   <time of dump>
*
*   Dumpfile:   <dumpfile name>
*
******************************************************************************
```

Figure 4-1. Analysis Header

The fields in Figure 4-1 are as follows:

<date of dump>	The date of the dump. You specified the data in this field when you invoked the Dumper.
<time of dump>	The time of the dump. You specified the data in this field when you invoked the Dumper.
<dumpfile name>	The name of the dump file. You specified this field when you invoked the Dumper.

See Chapter 3 for more information on the previous fields.

CURRENT PROCESSOR STATE

The section of the listing in Figure 4-2 displays the state of both the CPU running the system and the Numeric Processor Extension at the time you invoked the Dumper. If the registers of the processor are not available to the analyzer, it prints the message, "Registers not available" in place of the processor state.

```
%
%--------------------------------------------------------------------
%
%       Current processor state
%
%--------------------------------------------------------------------
%

   *
   *    CPU state
   *

   AX = <xxxx> SP = <xxxx> CS = <xxxx> IP = <xxxx>
   BX = <xxxx> BP = <xxxx> DS = <xxxx> FL = <Ox Dx Ix Tx Sx Zx Ax Px Cx>
   CX = <xxxx> SI = <xxxx> SS = <xxxx>
   DX = <xxxx> DI = <xxxx> ES = <xxxx>

   *
   *    NPX state
   *

   CW = <xxxx>        SW = <xxxx>      TW = <xxxx>
   IP = <xxxxx>       OC = <xxx>       OP = <xxxxx>
   ST(0) = <xxxxxxxxxxxxxxxxxxxxx>
   ST(1) = <xxxxxxxxxxxxxxxxxxxxx>
   ST(2) = <xxxxxxxxxxxxxxxxxxxxx>
   ST(3) = <xxxxxxxxxxxxxxxxxxxxx>
   ST(4) = <xxxxxxxxxxxxxxxxxxxxx>
   ST(5) = <xxxxxxxxxxxxxxxxxxxxx>
   ST(6) = <xxxxxxxxxxxxxxxxxxxxx>
   ST(7) = <xxxxxxxxxxxxxxxxxxxxx>
```

Figure 4-2. Current Processor State

The fields pertaining to the CPU in Figure 4-2 are as follows:

AX, SP, CS, IP, BX, BP, DS, FL, CX, SI, SS, DX, DI, ES	The hexadecimal values in the CPU's registers. The names of the processor's registers are as follows:

Name	Description
AX	"A" Register
SP	Stack Pointer
CS	Code Segment
IP	Instruction Pointer
BX	"B" Register
BP	Base Pointer
DS	Data Segment
FL	Flags
CX	"C" Register
SI	Source Index
SS	Stack Segment
DX	"D" Register
DI	Destination Index
ES	Extra Segment

The fields pertaining to the Numeric Processor Extension (NPX) in Figure 4-2 appear only if your system includes one.

CW, SW, TW, IP, OC, OP, — The hexadecimal value in the NPX's register. The names of the Numeric Processor Extension registers are as follows:

Name	Description
CW	Control Word
SW	Status Word
TW	Tag Word
IP	Instruction Pointer
OC	Operation Code
OP	Operand Pointer

ST(0) through ST(7) — The hexadecimal value of the NPX stack registers. The Analyzer displays these 80-bit registers in temporary real format.

JOB TREE

The section of the listing in Figure 4-3 displays the tokens of all the jobs in your system. The offspring jobs are indented to show their position in the hierarchy.

```
%
%----------------------------------------------------------------
%
%    iRMX 86 job tree
%
%----------------------------------------------------------------
%

          <xxxx1>
             <xxxx2>
                <xxxx3>
                   <xxxx4>
             <xxxx5>
             <xxxx6>
```

Figure 4-3. Job Tree

The fields in Figure 4-3 are as follows:

<xxxx$_1$>	The token for the root job.
<xxxx$_2$> through <xxxx$_6$>	The tokens for the offspring jobs of the root job. The offspring jobs are indented three spaces to show their position in the hierarchy.

LIST OF READY TASKS

The section of the listing in Figure 4-4 displays the token for the task that is running followed by the the tokens for the ready tasks.

```
%
%---------------------------------------------------------------
%
%    List of ready tasks
%
%---------------------------------------------------------------
%

                          Task                     Priority

         Running task     <xxxx>J/<yyyy>T          <aa>

         Ready tasks      <xxxx>J/<yyyy>T          <aa>
                              .
                              .
                              .
                          <xxxx>J/<yyyy>T          <aa>
```

Figure 4-4. List Of Ready Tasks

The fields in Figure 4-4 are as follows:

<xxxx>J	The token representing the job which contains the task.
<yyyy>T	The token representing either the running task or a ready task.
<aa>	The priority of the task.

Depending on what the processor was doing at the time of the dump, ROOTJ/IDLET can appear in either the running task or the ready task list. ROOTJ/IDLET represents the operating system's idle task; the idle task is a low-priority task that runs when no other tasks are running. If ROOTJ/DELET appears in the ready task list, a task requested deletion of itself or its job. See the iRMX 86 NUCLEUS REFERENCE MANUAL for more information about deleting a task or job.

LIST OF SLEEPING TASKS

The section of the listing in Figure 4-5 displays the token for the tasks that are sleeping. The sleeping tasks are shown in increasing order of their remaining sleep-time.

```
%
%-------------------------------------------------------------
%
%    List of sleeping tasks
%
%-------------------------------------------------------------
%

    Task                Priority      Delay          Delay
                                      remaining      requested

    <xxxx>J/<yyyy>T     <aa>          <bbbb>         <cccc>
          .
          .
          .
    <xxxx>J/<yyyy>T     <aa>          <bbbb>         <cccc>
```

Figure 4-5. List Of Sleeping Tasks

The fields in Figure 4-5 are as follows:

<xxxx>J	The token representing the job which contains the task.
<yyyy>T	The token representing the sleeping task.
<aa>	The priority of the task.
<bbbb>	The remaining time the task is required to sleep. This sleep-time is expressed in intervals of the system clock, so you must know the value of the clock interval in your system.
<cccc>	The sleep-time the task requested. This sleep-time is expressed in intervals of the system clock, so you must know the value of the clock interval in your system.

If ROOTJ/DELET appears in the list of sleeping tasks, your system was not deleting a task or a job at the time of the dump. See the iRMX 86 NUCLEUS REFERENCE MANUAL for more information about deleting a task or job.

If there are no tasks sleeping at the time of the dump, the Analyzer prints the sleeping task header and the message, "No tasks are sleeping."

LIST OF EXTENSIONS IN SYSTEM

The section of the listing in Figure 4-6 displays information about each extension in your system. It also shows the deletion mailbox for each extension along with its containing job.

```
%
%-------------------------------------------------------------
%
%    List of extensions
%
%-------------------------------------------------------------
%

        Extension      Extension      Containing     Deletion
        token          type           job            mailbox

        <uuuu>         <vvvv>         <wwww>J        <xxxx>J/<yyyy>M
          .
          .
          .
        <uuuu>         <vvvv>         <wwww>J        <xxxx>J/<yyyy>M
```

Figure 4-6. List Of Extensions In System

The fields in Figure 4-7 are as follows:

<uuuu>	The token for the extension.
<vvvv>	The WORD containing the type code for the new type. This type code was specified when the extension object was set up with system call CREATE$EXTENSION.
<wwww>J	The token for the job that contains the extension.
<xxxx>J	The token for the job that contains the deletion mailbox.
<yyyy>M	The token for the deletion mailbox set up with CREATE$EXTENSION.

LIST OF INTERRUPT TASKS

The section of the listing in Figure 4-7 displays information about your system's interrupt tasks in increasing order of interrupt level.

```
%
%------------------------------------------------------------
%
%    List of interrupt tasks
%
%------------------------------------------------------------
%

    Level          Task                      Data segment
                                             base

    <dd>           <xxxx>J/<yyyy>T           <zzzz>
      .
      .
      .
    <dd>           <xxxx>J/<yyyy>T           <zzzz>
```

Figure 4-7. List Of Interrupt Tasks

The fields in Figure 4-7 are as follows:

<dd>	A byte containing the interrupt level that the task services. The level is encoded as follows:

Bits	Value
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.

<xxxx>J	The token for the job that contains the interrupt task.
<yyyy>T	The token for the interrupt task.
<zzzz>	The token for the interrupt handler's data segment.

JOB REPORT ORGANIZATION

The Analyzer prints an entire job report for each job in your system. Because each job report consists of a number of detailed displays, this report is divided into sections as follows:

- Job Report Header
    - Information about the Job
    - Job Descriptor
- Object Directory
  Tasks Waiting for Object Lookup
- Objects Contained by Job
- Pool Report for Job
- Segments in Job
- Task Report
    - Non-Interrupt Tasks
    - Interrupt Tasks
- Mailbox Report
- Semaphore Report
- Region Report
- Extension Objects in Job
- Composites in Job
    - Extension Sub-Header
    - Composite Object Report
    - Special Composite Objects
        - Physical File Driver Connection Report
        - Stream File Driver Connection Report
        - Named File Driver Connection Report
        - Dynamic Device Information Report
        - Logical Device Object Report
        - I/O Job Object Report

JOB REPORT HEADER AND JOB INFORMATION

The section of the listing in Figure 4-8 contains the Job Report Header and the token of the job being printed. This header is followed by a "deletion pending message" and by information about the attributes of the job. Next is internal information about the job descriptor.

```
*
*****************************************************************
*****************************************************************
*
*    Job report, token = <xxxx>
*
*****************************************************************
*****************************************************************
*
   <deletion pending message>

 Current tasks   <xxxx>      Max tasks    <xxxx>      Max priority  <xx>
 Current objs    <xxxx>      Max objects  <xxxx>      Parameter obj <xxxx>
 Except handler  <xxxx:xxxx> Except mode  <xx>        Parent job    <xxxx>
 Job flags       <xxxx>

*
*    Job descriptor
*

   BBBB:0000  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>
   BBBB:0000  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>
   BBBB:0000  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>
   BBBB:0000  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>  <xxxx>
```

Figure 4-8. Job Report Header And Job Information

The fields in Figure 4-8 are as follows:

<deletion pending message>	This message is present only if there is some type of deletion pending against the job. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."
Current tasks	The number of tasks currently existing in the job.
Max tasks	The maximum number of tasks that can exist in the job at the same time. This value was set when the job was created with the system call RQ$CREATE$JOB.

Max priority	The maximum (lowest numerically) priority allowed for any task in the job. This value was set when the job was created with the system call RQ$CREATE$JOB.
Current objs	The number of objects currently existing in the job.
Max objects	The maximum number of objects that can exist in the job at the same time. This value was set when the job was created with the system call RQ$CREATE$JOB.
Parameter obj	The token for the object the parent job passed to this job. This value was set when the job was created with the system call RQ$CREATE$JOB.
Except handler	The start address of the job's exception handler. This address was set when the job was created with the system call RQ$CREATE$JOB.
Except mode	The value that indicates when control is to be passed to the new job's exception handler. It is encoded as follows:

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

This value was set when the job was created with the system call RQ$CREATE$JOB.

Parent job	The token for the parent job of this job.
Job flags	The job flags parameter that was specified when the job was created. The bits (where bit 15 is the high-order bit) have the following meanings:

Bit	Meaning
15-2	Reserved.
1	If 0, then whenever a task in the new job or any of its descendent jobs makes a Nucleus system call, the Nucleus checks the parameters for validity.

	If 1, the Nucleus does not check the parameters of Nucleus system calls made by tasks in the new job. However, if any offspring of the new job has been created with this bit set to 0, there will be parameter checking for the new job.
0	Reserved.

The job descriptor has information which is not useful to application and system programmers. You should ignore this information.

OBJECT DIRECTORY AND TASKS WAITING FOR OBJECT LOOKUP

The section of the listing in Figure 4-9 displays the names and tokens for the objects cataloged in the job's object directory. This information is followed the uncataloged names of the objects for which a task is waiting. For each such object, the Analyzer displays the requested name, the token for the task, and the token for the containing job of the first task waiting for object lookup.

```
*
*    Object directory
*

   Maximum entries: <aaaa>  Entries used: <bbbb>

   Name             Length    Hex representation           Object
                    in hex

   <name1>          <z>       <xx xx xx xx...>             <xxxx>J/<yyyy>t
     .
   <namen>          <z>       <xx xx xx xx...>             <xxxx>J/<yyyy>t
*
*    Tasks waiting for object lookup
*

   Name             Length    Hex representation           Task
   requested        in hex

   <namea>          <z>       <xx xx xx xx...>             <xxxx>J/<yyyy>T
     .
   <namez>          <z>       <xx xx xx xx...>             <xxxx>J/<yyyy>T
```

Figure 4-9. Object Directory And Tasks Waiting For Object Lookup

The fields in Figure 4-9 are as follows:

Maximum entries	The maximum allowable number of entries this job can have in its object directory.
Entries used	The number of entries used within the directory.
$<name_1>$ through $<name_n>$	The names under which the objects are cataloged. The printable characters are shown for each name in the list. Characters which cannot be printed are replaced with a period (.) in this listing.
<z>	The length of the name in bytes.

<xx xx xx xx...>	The hexadecimal representation of each letter in the name.
<xxxx>J	The token for the job that contains the object.
<yyyy>t	The token for the object where "t" is one of the following characters that identify iRMX 86 object types:

Character	Object Type
C	composite
G	segment
J	job
M	mailbox
R	region
S	semaphore
T	task
X	extension

The fields pertaining to the tasks waiting for object lookup in Figure 4-9 are as follows:

<name$_a$> through <name$_z$>	The name of the object the task has requested.
<z>	The length of the name in bytes.
<xx xx xx xx...>	The hexadecimal representation of each letter in the name.
<xxxx>J	The token for the job that contains the task.
<yyyy>T	The token for the task.

OBJECTS CONTAINED BY JOB

The section of the listing in Figure 4-10 lists the tokens for a job's child jobs, task, mailboxes, semaphores, regions, segments, extensions, and composites.

```
 *
 *     Objects contained by job <xxxx>
 *

       Child jobs:   <xxxx>  <xxxx>  <xxxx>... <xxxx>
       Tasks:        <xxxx>  <xxxx>  <xxxx>... <xxxx>
       Mailboxes:    <xxxx>  <xxxx>  <xxxx>... <xxxx>
       Semaphores:   <xxxx>  <xxxx>  <xxxx>... <xxxx>
       Regions:      <xxxx>  <xxxx>  <xxxx>... <xxxx>
       Segments:     <xxxx>  <xxxx>  <xxxx>... <xxxx>
       Extensions:   <xxxx>  <xxxx>  <xxxx>... <xxxx>
       Composites:   <xxxx>  <xxxx>  <xxxx>... <xxxx>
```

Figure 4-10. Objects Contained By Job

The fields in Figure 4-10 are as follows:

Child jobs	The tokens for the child jobs within the job.
Tasks	The tokens for the tasks within the job.
Mailboxes	The tokens for the mailboxes within the job. A lower-case "o" immediately following a token for a mailbox means that one or more objects are queued at the mailbox. A lower-case "t" immediately following a token for a mailbox means that one or more tasks are queued at the mailbox.
Semaphores	The tokens for all the semaphores within the job. A lower-case "t" immediately following a token for a semaphore means that one or more tasks are queued at the semaphore.
Regions	The tokens for all the regions within the job. A lower-case "b" (busy) immediately following a token for a region means that a task is accessing information guarded by the region.
Segments	The tokens for all the segments within the job.
Extensions	The tokens for all the extensions within the job.
Composites	The tokens for all the composites within the job.

POOL REPORT

The section of the listing in Figure 4-11 displays information about the job's memory pool. It also shows the base address and size of any unallocated memory areas.

```
%
%-----------------------------------------------------------------------
%
%     Pool report for job xxxx
%
%-----------------------------------------------------------------------
%

      Pool min     <xxxx>    Pool Max     <xxxx>    Initial size  <xxxx>
      Pool size    <xxxx>    Largest seg  <xxxx>

*
*     Available pool memory areas
*

      Base                   Size

      <BBBB>                 <ssss>
        .
        .
        .
      <BBBB>                 <ssss>

      Total available        <xxxx>
      Total allocated        <xxxx>
```

Figure 4-11. Pool Report

The fields in Figure 4-11 are as follows:

Pool min	The minimum size (in 16-byte paragraphs) of the job's memory pool. This value was set when the job was created.
Pool max	The maximum size (in 16-byte paragraphs) of the job's memory pool. This value was set when the job was created.
Initial size	The initial size (in 16-byte paragraphs) of the job's memory pool.
Pool size	The current size (in 16-byte paragraphs) of the job's memory pool.
Largest seg	The number of 16-byte paragraphs in the largest segment in the job's memory pool.

The fields pertaining to the available pool memory areas in Figure 4-11 are as follows:

<BBBB>	The base address of the unallocated memory area. Each memory area is located at an offset of 0 from the given base.
<ssss>	The size of the unallocated memory area in 16-byte paragraphs.
Total available	The total amount of unallocated memory in 16-byte paragraphs.
Total allocated	The total amount of allocated memory in 16-byte paragraphs.

SEGMENTS IN JOB

The section of the listing in Figure 4-12 displays information about the segments in the pool of the job. It displays the token and the size of the segment along with the contents of the segment.

```
%
%--------------------------------------------------------------------
%
%     Segments in job <xxxx>
%
%--------------------------------------------------------------------
%

    Segment       Size
    token

    <xxxx>        <yyyy>        descriptor: BBBB:0000 <xxxx> <xxxx>...
    <deletion pending message>
                                contents:   BBBB:0000 <xx> <xx>...*a...a*
      .
      .
      .
    <xxxx>        <yyyy>        descriptor: BBBB:0000 <xxxx> <xxxx>...
                                contents:   BBBB:0000 <xx> <xx>...*a...a*
```

Figure 4-12. Segments In Job

The fields in Figure 4-12 are as follows:

<deletion pending message>	This message is present only if there is some type of deletion pending against the object. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."
<xxxx>	The token for the segment.
<yyyy>	The size of the segment in 16-byte paragraphs.
descriptor	The segment descriptor contains information which is not useful to application and system programmers. You should ignore this information.
contents	If you specified the BYTE option when you invoked the Analyzer, the contents of the segment will be displayed in byte format as shown in Figure 4-13.

BBBB:OOOO	The base and offset address of the segment.
<xx>	A pair of hexadecimal digits representing a byte.
a	The ASCII representation of the corresponding byte (if printable). If the byte cannot be printed, the Analyzer places a period (.) in its place.

If you specified the WORD option, the contents of the segment is displayed in WORD format with 8 words to a line. The ASCII representation <*a...a*> is not displayed in the WORD format.

If you did not specify the BYTE or the WORD option when you invoked the Analyzer, the contents display does not appear.

If you specified both the BYTE and WORD option when you invoked the Analyzer, the contents field appears in both formats.

TASK REPORT

The Analyzer lists information about tasks in two different ways. Figure 4-13 shows the format for a non-interrupt task and Figure 4-14 shows the format for an interrupt task.

```
%
%-----------------------------------------------------------------------
%
%  Task report, token = <xxxx>
%
%-----------------------------------------------------------------------
%

   <deletion pending message>

  Static pri      <xx>            Dynamic pri      <xx>    Task state      <xxxx>
  Suspend depth   <xx>            Delay req        <xxxx>  Last exchange   <xxxx>
  Except handler  <xxxx:xxxx>     Except mode      <xx>    Task flags      <xx>
  Containing job  <xxxx>          Interrupt task    no

*
*  Task descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
        .
        .
        .
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>

*
*  Task stack segment
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>...
```

Figure 4-13. Non-Interrupt Task Report

```
%
%-----------------------------------------------------------------------
%
%  Task report, token = <xxxx>
%
%-----------------------------------------------------------------------

   <deletion pending message>

 Static pri       <xx>         Dynamic pri       <xx>    Task state      <xxxx>
 Suspend depth    <xx>         Delay req         <xxxx>  Last exchange   <xxxx>
 Except handler   <xxxx:xxxx>  Except mode       <xx>    Task flags      <xx>
 Containing job   <xxxx>       Interrupt task     yes    Int level       <xx>
 Pending int      <xx>         Max interrupts    <xx>    Master mask     <xx>
 Slave mask       <xx>         Slave number      <xx>

*
*  Task descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
        .
        .
        .
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>


*
*  Task stack segment
*

   Tasks  SS:SP   xxxx:xxxx

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>...
```

Figure 4-14. Interrupt Task Report

The fields in Figure 4-13 and 4-14 are as follows:

<deletion pending message>	This message is present only if there is some type of deletion pending against the object. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."
Static pri	The current priority of the task. This value was set when the job was created with the system call RQ$CREATE$TASK.

Dynamic pri	A temporary priority that the Nucleus sometimes assigns to the task (temporarily) in order to improve system performance.
Task state	The state of the task. There are five possible states:

State	Description
ready	ready for execution
asleep	task is asleep
suspended	task is suspended
asleep/susp	task is both asleep and suspended
deleted	task is being deleted

If this field can't be interpreted, the Analyzer displays the actual hexadecimal value followed by a space and two question marks.

Suspend depth	The current number of outstanding RQ$SUSPEND$TASK system calls applied to this task without corresponding RQ$RESUME$TASK system calls.
Delay req	The number of sleep units the task requested. See the iRMX 86 NUCLEUS REFERENCE MANUAL for more information on sleep units.
Last exchange	The token for the mailbox, region, or semaphore at which the task is currently waiting.
Except handler	The start address of the task's exception handler. This value was set when the task was created with RQ$CREATE$TASK, RQ$CREATE$JOB, or RQ$CREATE$IO$JOB, or when RQ$SET$EXCEPTION$HANDLER was used.
Except mode	The value used to indicate when control is to be passed to the task's exception handler. It is encoded as follows:

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

This value was set when the task was created with RQ$CREATE$TASK, RQ$CREATE$JOB, or RQ$CREATE$IO$JOB, or when RQ$SET$EXCEPTION$HANDLER was used.

Task flags	The task flags parameter used when the task was created with the system call RQ$CREATE$TASK. The bits (where 15 is the high-order bit) have the following meanings:

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

Containing job	The token for the job which contains this task.
Interrupt task	"No" signifies that the task is not an interrupt task. In this case, there are no more fields in the display (see Figure 4-14). "Yes" signifies that the task is an interrupt task. In this case, there are six more fields in the display (see Figure 4-15).
Int level	The interrupt level that the interrupt task services. This level was set when the system call RQ$SET$INTERRUPT was used.
Pending int	The number of RQ$SIGNAL$INTERRUPT calls that are pending.
Max interrupts	The maximum number of RQ$SIGNAL$INTERRUPT calls that can be pending.
Master mask	The hexadecimal value associated with the interrupt mask for the master interrupt controller. This value comes from the bits that correspond to the different master interrupt levels. Remember that bit numbers corresponds to interrupt level numbers. For example, bit 0 corresponds to interrupt level 0 and bit 7 corresponds to interrupt level 7. If the bit is set, the corresponding interrupt is disabled. For more information see the iRMX 86 NUCLEUS REFERENCE MANUAL.

Slave mask	The hexadecimal value associated with the interrupt mask for a slave interrupt controller. This value comes from the bits that correspond to the different slave interrupt levels. Remember that bit numbers correspond to interrupt level numbers. For example, bit 0 corresponds to interrupt level 0 and bit 7 corresponds to interrupt level 7. If the bit is set, the corresponding interrupt is disabled. For more information see the iRMX 86 NUCLEUS REFERENCE MANUAL.
Slave number	The programmable interrupt controller number of the slave that is referred to by the slave mask. For more information see the iRMX 86 NUCLEUS REFERENCE MANUAL.

The task descriptor has information which is not useful to application and system programmers. You should ignore this information.

The task stack segment displays the address of the stack segment:stack pointer (SS:SP) along with a hexadecimal display of the contents of the task's stack segment beginning at SS:SP. The task's stack segment contains part of the data in your task beginning at SS:SP.

MAILBOX REPORT

The Analyzer lists information about mailboxes in three different ways. The first listing (Figure 4-15) appears when nothing is queued at the mailbox, the second listing (Figure 4-16) appears when objects are queued at the mailbox, and the third listing (Figure 4-17) appears when tasks are queued at the mailbox.

```
%
%-----------------------------------------------------------------------
%
%  Mailbox report, token = <xxxx>
%
%-----------------------------------------------------------------------
%

   <deletion pending message>

  Containing job      <xxxx>                  Queue discipline     <xxxx>
  Task queue head     <xxxx>                  Object queue head    <xxxx>
  Object cache depth  <xx>

*
*  Mailbox descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
```

Figure 4-15. Mailbox Report (Mailbox With No Queue)

```
%
%-------------------------------------------------------------------------
%
%  Mailbox report, token = <xxxx>
%
%-------------------------------------------------------------------------
%

   <deletion pending message>

  Containing job      <xxxx>                Queue discipline     <xxxx>
  Task queue head     <xxxx>                Object queue head    <xxxx>
  Object cache depth  <xxxx>

  Object queue   <xxxx>J/<yyyy>t     <xxxx>J/<yyyy>t     <xxxx>J/<yyyy>t...

*
*  Mailbox descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
```

Figure 4-16. Mailbox Report (Mailbox With Object Queue)

```
%
%-------------------------------------------------------------------------
%
%  Mailbox report, token = <xxxx>
%
%-------------------------------------------------------------------------
%

   <deletion pending message>

  Containing job      <xxxx>                Queue discipline     <xxxx>
  Task queue head     <xxxx>                Object queue head    <xxxx>
  Object cache depth  <xxxx>

  Task queue        <xxxx>J/<yyyy>T   <xxxx>J/<yyyy>T   <xxxx>J/<yyyy>T...

*
*  Mailbox descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
```

Figure 4-17. Mailbox Report (Mailbox With Task Queue)

The fields in Figures 4-15, 4-16, and 4-17 are as follows:

<deletion pending message>	This message is present only if there is some type of deletion pending against the object. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."
Containing job	The token for the job that contains this mailbox.
Queue discipline	The order in which you specified the tasks (making requests from the mailbox) be queued. This order is set up with RQ$CREATE$MAILBOX. The tasks can be order in a "first-in/first-out" (FIFO) method or in a priority-based method (PRI).
Task queue head	The token for the task at the head of the queue.
Object queue head	The token for the object at the head of the queue.
Object cache depth	The maximum number of entries allowed in the high-performance queue associated with the mailbox. The size of this cache was set up when the mailbox was created with RQ$CREATE$MAILBOX. When the list of tokens in the object queue is greater than the object cache depth, you have temporarily overflowed your high-performance queue. Succeeding objects are stored in a low-performance queue associated with the mailbox.
Object queue	A list of tokens for the objects queued at the mailbox and their containing jobs where:

<xxxx>J The token for the job that contains the object.

<yyyy>t The token for the object where "t" is one of the following characters that identify iRMX 86 object types:

Character	Object Type
C	composite
G	segment
J	job
M	mailbox
R	region
S	semaphore
T	task
X	extension

This list appears in the display only if there are objects queued at the mailbox.

Task queue	A list of tokens for the tasks queued at the mailbox and their containing jobs where:

<xxxx>J	The token for the job that contains the task.
<yyyy>T	The token for the task.

This list appears in the display only if there are tasks queued at the mailbox.

The mailbox descriptor contains information which is not useful to system and application engineers. You should ignore this information.

SEMAPHORE REPORT

The Analyzer lists information about semaphores in two ways. The first listing (Figure 4-18) appears when no tasks are queued at the semaphore, and the second listing (Figure 4-19) appears when tasks are queued at the semaphore.

```
%
%-----------------------------------------------------------------------
%
%  Semaphore report, token = <xxxx>
%
%-----------------------------------------------------------------------
%

    <deletion pending message>

   Containing job      <xxxx>                    Queue discipline    <xxxx>
   Task queue head     <xxxx>                    Maximum value       <xxxx>
   Current value       <xxxx>

*
*  Semaphore descriptor
*

    BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
    BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
```

Figure 4-18. Semaphore Report (Semaphore With No Queue)

```
%
%----------------------------------------------------------------------
%
%  Semaphore report, token = <xxxx>
%
%----------------------------------------------------------------------
%

   <deletion pending message>

  Containing job     <xxxx>                 Queue discipline    <xxxx>
  Task queue head    <xxxx>                 Maximum value       <xxxx>
  Current value      <xxxx>

  Task queue       <xxxx>J/<yyyy>T  <xxxx>J/<yyyy>T   <xxxx>J/<yyyy>T...

*
*  Semaphore descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
```

Figure 4-19. Semaphore Report (Semaphore With Task Queue)

The fields in Figures 4-18 and 4-19 are as follows:

<deletion pending message>	This message is present only if there is some type of deletion pending against the object. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."
Containing job	The token for the job which contains the semaphore.
Queue discipline	The way the tasks are ordered in the queue. The tasks can be ordered in a "first-in/first-out" (FIFO) method or a priority based method (PRI) when the semaphore is created with RQ$CREATE$SEMAPHORE.
Task queue head	The token for the task at the head of the queue.
Maximum value	The maximum number of units the semaphore can have. This number was set when the semaphore was created with RQ$CREATE$SEMAPHORE.
Current value	The number of units currently contained in the semaphore.

Task queue	A list of tokens for the tasks queued at the semaphore and their containing jobs where:

<xxxx>J	The token for the job that contains the task.
<yyyy>T	The token for the task.

This list appears in the display only if there are tasks queued at the semaphore.

The semaphore descriptor has information which is not useful to application and system programmers. You should ignore this information.

REGION REPORT

The Analyzer lists information about regions in two ways. The first listing (Figure 4-20) appears when no tasks are queued at the region, and the second listing (Figure 4-21) appears when tasks are queued at the region.

```
%
%---------------------------------------------------------------------
%
%  Region report, token = <xxxx>
%
%---------------------------------------------------------------------
%

   <deletion pending message>

   Containing Job    <xxxx>       Queue discipline     <xxxx>
   Entered task      <xxxx>

*
*  Region descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
```

Figure 4-20. Region Report (Region With No Queue)

```
%
%---------------------------------------------------------------------
%
%  Region report, token = <xxxx>
%
%---------------------------------------------------------------------
%

   <deletion pending message>

   Containing Job    <xxxx>       Queue discipline     <xxxx>
   Entered task      <xxxx>

   Task queue       <xxxx>J/<yyyy>T  <xxxx>J/<yyyy>T   <xxxx>J/<yyyy>T...

*
*  Region descriptor
*

   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
   BBBB:0000 <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx> <xxxx>
```

Figure 4-21. Region Report (Region With Task Queue)

The fields in Figures 4-20 and 4-21 are as follows:

<deletion pending message>	This message is present only if there is some type of deletion pending against the object. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."
Containing job	The token for the job that contains the region.
Queue discipline	The way you ordered the tasks in the queue. The tasks can be ordered in a "first-in/first-out" (FIFO) method or in a priority-based method (PRI) when the region is created with RQ$CREATE$REGION.
Entered task	The token for the task that is currently accessing information guarded by the region.
Task queue	A list of tokens for the tasks queued at the region and their containing jobs where: <xxxx>J The token for the job that contains the task. <yyyy>T The token for the task. This list appears in the display only if there are tasks queued at the region.

The region descriptor contains information which is not useful to application and system Engineers. You should ignore this information.

EXTENSION OBJECTS IN JOB

This section of the listing displays the tokens for all of the extension objects contained by the job as shown in Figure 4-22. It then displays information about each extension along with its descriptor.

```
%
%------------------------------------------------------------------
%
%  Extension objects in job <xxxx>
%
%------------------------------------------------------------------

   Token  Extension  Deletion
          type       mailbox

   <aaaa>  <bbbb>    <cccc>      descriptor:       BBBB:0000 <xxxx> <xxxx>...
   <deletion pending message>                      BBBB:0000<xxxx>xxxx...
                                 composite list:   <xxxx>J/<yyyy>X
                                                   <xxxx>J/<yyyy>X...
      .
      .
      .
   <aaaa>  <bbbb>    <cccc>      descriptor:       BBBB:0000<xxxx>xxxx...
                                                   BBBB:0000<xxxx>xxxx...
                                 composite list:   <xxxx>J/<yyyy>X
                                                   <xxxx>J/<yyyy>X...
```

Figure 4-22. Extension List

The fields in Figure 4-22 are as follows:

<aaaa>	The token for the extension object.
<bbbb>	The extension type code for the extension. This code was specified when the extension was created with RQ$CREATE$EXTENSION. This extension object represents the license to create composite objects of this type./
<cccc>	The token for the mailbox to which this extension goes when it is to be deleted. This mailbox was specified when the extension was created with RQ$CREATE$EXTENSION.
<deletion pending message>	This message is present only if there is some type of deletion pending against the object. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."

The extension descriptor contains information which is not useful to application and system Engineers. You should ignore this information.

The composite list consists of a list of composite tokens and the jobs that contain the tokens for the objects of this extension type, where:

<xxxx>J	The token for the job that contains the object.
<yyyy>X	The token for the object where "X" identifies the token as an extension.

COMPOSITE LIST REPORT

If the job contains any composite objects, the Analyzer displays a Composite List Report. The Composite List Report consists of the following sections:

- Composite List Report Header
- Extension Sub-Header
- Composite Object Report

The Composite List Report contains a composite list report header followed by one extension sub-header (Figure 4-23) for each extension type with composite objects in the job. Each extension sub-header consists of information about the extension type, the extension object, the extension's containing job, and the deletion mailbox for the extension object.

Each extension sub-header is followed by a list of Composite Object Reports. The Analyzer displays either a general composite object report or one of six special reports for Basic I/O System (BIOS) composites. The types of reports for BIOS composites are as follows:

- Physical File Driver Connection Report
- Steam File Driver Connection Report
- Named File Driver Connection Report
- Dynamic Device Information Report
- Logical Device Object Report
- I/O Job Object Report

Each of the special reports contain information from the general composite object report along with information special to the specific composite. Because some fields shown in the figures in this section are repeated, this manual avoids unnecessary repetition by explaining only those fields introduced in the figure.

COMPOSITE LIST REPORT HEADER AND EXTENSION SUB-HEADER

Figure 4-23 shows the composite list report header followed by an extension sub-header. Each sub-header contains general information concerning the extension object.

NOTE

> Remember that the extension object can be contained in a different job than the one that contains the composite object. You should refer to the Extension Report in the extension's containing job for more detailed information on the extension object.

```
%
%-----------------------------------------------------------------------
%
%  Composites in job <xxxx>
%
%-----------------------------------------------------------------------
%


*
*  Extension type                <xxxx>
*  Extension object              <xxxx>
*  Extensions containing job     <xxxx>
*  Deletion mailbox              <xxxx>
*
```

Figure 4-23. Composite List Report Header And Extension Sub-Header

The fields in the extension's subheader (Figure 4-23) are as follows:

Extension type	The extension type code for the composite. This code was specified when the composite was created with RQ$CREATE$COMPOSITE.
Extension object	The token for the extension object that represents the license to create this type of composite.
Extensions containing job	The token for the job that contains the composite.
Deletion mailbox	The token for the mailbox to which this composite goes when it is to be deleted. This mailbox was specified when the extension was created with RQ$CREATE$EXTENSION.

GENERAL COMPOSITE OBJECT REPORT

Figure 4-24 shows the composite object report for all composites except special composites. Special composites include Physical File Driver Connection reports, Stream File Driver Connection reports, Named File Driver Connection reports, Dynamic Device Information, Logical Device Information, and I/O Job Object reports. These special composites displays appear in place of the general composite object report.

```
 *
 *  Composite object, token = <xxxx>
 *

    <deletion pending message>
    Extension type      <xxxx>    descriptor:  BBBB:0000 <xxxx> <xxxx>...
                                               BBBB:0000 <xxxx> <xxxx>...
                                               BBBB:0000 <xxxx> <xxxx>...
    Number of slots     <xxxx>
    Object size         <xxxx>
    Component List      <xxxx>J/<yyyy>t   <xxxx>J/<yyyy>t   <xxxx>J/<yyyy>t...
```

Figure 4-24. General Composite Object Report

The fields in Figure 4-24 are as follows:

<deletion pending message>	This message is present only if there is some type of deletion pending against the object. The messages are either, "DELETION PENDING" or "FORCED DELETION PENDING."
Extension type	The extension type code for the composite. This code was specified when the composite was created with RQ$CREATE$COMPOSITE.
Number of slots	The number of positions available in the composite for tokens of component objects. This value was set when the composite was created with RQ$CREATE$COMPOSITE.
Object size	The size of the object in paragraphs.

The descriptor contains information which is not useful to application and system Engineers. You should ignore this information.

The component list consists of a list of tokens and their containing jobs for the objects that currently make up the composite, where:

<xxxx>J	The token for the job that contains the object.
<yyyy>t	The token for the object where "t" is one of the following characters that identify iRMX 86 object types:

Character	Object Type
C	composite
G	segment
J	job
M	mailbox
R	region
S	semaphore
T	task
X	extension

PHYSICAL FILE DRIVER CONNECTION REPORT

Figure 4-25 shows the listing for a connection to a physical file.

```
*
*  Physical file driver connection, token = <xxxx>
*

  <deletion pending message>
  Extension type  <xxxx>   descriptor:  BBBB:0000 <xxxx> <xxxx> <xxxx>...
                                        BBBB:0000 <xxxx> <xxxx> <xxxx>...
                                        BBBB:0000 <xxxx> <xxxx> <xxxx>...
  Containing job  <xxxx>   Conn flags     <xx>       Access          <xxxx>
  Open mode       <xxxx>   Open share     <xxxx>     File pointer    <xxxx:xxxx>
  File node       <xxxx>   Device desc    <xxxx>     DUIB pointer    <xxxx:xxxx>
  Num of conn     <xxxx>   Num of readers <xxxx>     Num of writers  <xxxx>
  File type       <xxxx>   File share     <xxxx>     Device conn     <xxxx>
```

Figure 4-25. Physical File Driver Connection Report

The fields introduced in Figure 4-25 are as follows:

Conn flags	The flags for the connection. The connection is active if bit 1 is set to one; the connection is a device connection if bit 2 is set to one.
Access	The access rights for this connection. The access rights are displayed in the same format as the display access rights for the DIR command in the Human Interface. This display uses a single character to represent a particular access right. If the file has the access right, the character appears. However, if the file does not have the access right, a dash (-) appears in the character position. The access rights along with the characters that represent them are as follows:

```
                    |-------- Delete
                    |r------- List
Directory files:    ||r------ Add
                    |||r----- Change
                    DLAC

                    DRAU
                    |||L----- Update
Data Files:         ||L------ Append
                    |L------- Read
                    L-------- Delete
```

Open mode	The mode established when this connection was opened. The possible values are:

Open Mode	Description
Closed	Connection is closed
Read	Connection is open for reading
Write	Connection is open for writing
R/W	Connection is open for reading and writing

If this field can't be interpreted, the Analyzer displays the actual hexadecimal value followed by a space and two question marks. This value is set during a RQSOPEN or RQAOPEN system call. See the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL or the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

Open share	The sharing status established when this connection was opened. The possible values are:

Share Mode	Description
Private	Private use only
Readers	File can be shared with readers
Writers	File can be shared with writers
ALL	File can be shared with all users

If this field can't be interpreted, the Analyzer displays the actual hexadecimal value followed by a space and two question marks. This value is set during an RQSOPEN or an RQAOPEN system call. See the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL or the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

File pointer	The current contents of the file pointer for this connection.
File node	A token for a segment that the Operating System uses to maintain information about the connection. The information in this segment appears in the next two fields.
Device desc	A token for the segment that contains the device descriptor. The device descriptor is used by the Operating System to maintain information about the connections to the device.
DUIB pointer	The address of the Device Unit Information Block (DUIB). See the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O OPERATING SYSTEMS for more information on the DUIB.

Num of conn	The number of connections to the file.
Num of readers	The number of connections currently open for reading.
Num of writers	The number of connections currently open for writing.
File type	The type of file. This field is for Named files only so it does not apply (N/A) to this display.
File share	The share mode of the file. This parameter defines how the file can be opened. The possible values are:

Share Mode	Description
Private	Private use only
Readers	File can be shared with readers
Writers	File can be shared with writers
ALL	File can be shared with all users

If this field can't be interpreted, the Analyzer displays the actual hexadecimal value followed by a space and two question marks. This value is set during RQSOPEN or RQAOPEN system calls. See the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL or the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

Device conn	The number of connections to the device.

STREAM FILE DRIVER CONNECTION REPORT

Figure 4-26 shows the listing for a stream connection.

```
*
*  Stream file driver connection, token = <xxxx>
*

   <deletion pending message>
   Extension type   <xxxx>  descriptor:  BBBB:0000 <xxxx> <xxxx> <xxxx>...
                                         BBBB:0000 <xxxx> <xxxx> <xxxx>...
   Containing job   <xxxx>  Conn flags      <xx>      Access         <xxxx>
   Open mode        <xxxx>  Open share      <xxxx>    File pointer   <xxxxxxxx>
   File node        <xxxx>  Device desc     <xxxx>    DUIB pointer   <xxxx:xxxx>
   Num of conn      <xxxx>  Num of readers  <xxxx>    Num of writers <xxxx>
   File type        <xxxx>  File share      <xxxx>    Device conn    <xxxx>
   Req queued       <xxxx>  Queued conn     <xxxx>    Open conn      <xxxx>
```

Figure 4-26. Stream File Driver Connection Report

The fields introduced in Figure 4-26 are as follows:

Req queued	The number of requests that are currently queued at the stream file.
Queued conn	The number of connections that are currently queued at the stream file.
Open conn	The number of connections that are currently open on the stream file.

See the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about the previous fields.

NAMED FILE DRIVER CONNECTION REPORT

Figure 4-27 shows the listing for a named file connection.

```
 *
 *  Named file driver connection, token = <xxxx>
 *

  <deletion pending message>
  Extension type <xxxx>     descriptor:  BBBB:0000 <xxxx> <xxxx> <xxxx>...
                                         BBBB:0000 <xxxx> <xxxx> <xxxx>...
  Containing job <xxxx>     Conn flags     <xx>       Access         <xxxx>
  Open mode      <xxxx>     Open share     <xxxx>     File pointer   xxxxxxxx
  File node      <xxxx>     Device desc    <xxxx>     DUIB pointer   <xxxx:xxxx>
  Num of conn    <xxxx>     Num of readers <xxxx>     Num of writers <xxxx>
  File type      <xxxx>     File share     <xxxx>     Device conn    <xxxx>
  Fnode flags    <xxxx>     Owner          <xxxx>     File ID        <xxxx>
  File gran      <xxxx>     Fnode ptr(s)   <xxxx:xxxx> Total blocks  <xxxxxxxx>
  Alloc size     <xxxxxxxx> File size      <xxxxxxxx> Volume name    <xxxxxx>
  Volume gran    <xxxx>     Volume size    <xxxxxxxx>
```

Figure 4-27. Named File Driver Connection Report

The fields introduced in Figure 4-27 are as follows:

File type — The type of file. The possible values are:

File Type	Description
DIR	Directory file
DATA	Data file

Fnode flags — A word containing flag bits. Each bit has a corresponding description. If that bit is one, then the corresponding description is true; if the bit is zero, then the corresponding description is false.

Bit	Description
0	This fnode is allocated
1	The file is a long file
2	Primary fnode
3-4	Reserved
5	This file has been modified
6	This file is marked for deletion
7-15	Reserved

Owner	The ID of the owner of the file. If this field has a value of FFFF, then the field is interpreted as "WORLD." See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
File ID	The number of the file's fnode. The fnode is a Basic I/O System data structure containing file attribute and status data.
File gran	The granularity of the file (in volume granularity units).
Fnode ptr(s)	The values of the fnode pointers. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
Total blocks	The total number of volume blocks currently used for the file; this includes indirect blocks. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
Alloc size	The total size (in bytes) allocated to the file. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
File size	The size (in bytes) of the file. See the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
Volume name	The name of the volume.
Volume gran	The granularity (in bytes) of the volume.
Volume size	The size (in bytes) of the volume.

DYNAMIC DEVICE INFORMATION REPORT

Figure 4-28 shows the information the Analyzer displays when a file has a dynamically created Device Unit Information Block (DUIB).

```
*
*  Dynamic device information for connection  <xxxx>
*

  File drivers    <xxxx>    Device gran    <xxxx>     Device size    <xxxx>
  Device functs   <xxxx>    Device name    <xxxx>
```

Figure 4-28. Dynamic Device Information Report

The fields introduced in Figure 4-28 are as follows:

File drivers — The validity of the file driver. The bits are associated with the file drivers as follows:

Bit	File Driver
0	physical
1	stream
3	named

Device gran — The value of the the volume granularity specified when the volume was formatted.

Device size — The number of bytes of information that the device-unit can store.

Device functs — The I/O function validity for this device-unit. The bits associated with the functions as follows:

Bit	Function
0	READ
1	WRITE
2	SEEK
3	SPECIAL
4	ATTACH DEVICE
5	DETACH DEVICE
6	OPEN
7	CLOSE

Device name — The name of the DUIB.

See the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM for more information concerning the previous fields.

LOGICAL DEVICE OBJECT REPORT

The listing in Figure 4-29 shows the device names and system logical names of the logical device composite object.

```
*
*  Logical device object, token = <xxxx>
*
   <deletion pending message>
  Extension type  <xxxx>    descriptor  BBBB:0000  xxxx  xxxx  xxxx...
                                         BBBB:0000  xxxx  xxxx  xxxx...
  Containing job  <xxxx>    Physical conn  <xxxx>    File driver    xx
  Owner ID        <xxxx>

                            Name           Length    Hex representation
  Device name               <aaaaaaa>      <bb>      <xx xx xx xx xx xx...>
  Sys logical name(s)       <aaaaaaa>      <bb>      <xx xx xx xx xx xx...>
```

Figure 4-29. Logical Device Object Report

The fields introduced in Figure 4-29 are as follows:

Physical conn	The token for the physical connection.
Device Name	The 1-to 14-character name under which the logical device object is cataloged. This name was specified when RQ$LOGICAL$ATTACH$DEVICE was called.
Sys logical name(s)	The 1-to 14-character name under which the the system logical name is cataloged. This name was specified when RQ$LOGICAL$ATTACH$DEVICE was called.
<bb>	The length of the device name or the system logical name. This name was specified in the DUIB during Basic I/O System configuration.
<xx>	The hexadecimal representation of each letter in the device name or the system logical name.

I/O JOB OBJECT REPORT

The section of the listing in Figure 4-30 displays information about exit messages in I/O job objects.

```
*
*  I/O job object, token = <xxxx>
*

  Extension type      <xxxx>    descriptor:  BBBB:0000  <xxxx>  <xxxx>...
                                             BBBB:0000  <xxxx>  <xxxx>...
  Exit message token  <xxxx>
  Exit message mbx    <xxxx>
```

Figure 4-30. I/O Job Object Report

The fields introduced in Figure 4-30 are as follows:

Exit message token	The token for the segment containing the exit message.
Exit message mbx	The mailbox that contains the exit message segment.

See the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about the previous fields.

SUMMARY OF ERRORS

Figure 4-31 shows the header for the summary of errors. This summary lists all error messages that the Analyzer encountered during the analysis. The summary also includes the page number of the listing on which the error occurred. For more information on error messages and their meanings, see the section in this chapter entitled, "Error Messages."

```
%
%-----------------------------------------------------------------------
%
%  Summary of errors detected by analysis
%
%-----------------------------------------------------------------------
%
   error message                              Page xx
        .                                        .
        .                                        .
        .                                        .
   error message                              Page xx
```

Figure 4-31. Summary Of Errors

The fields in Figure 4-31 are as follows:

error message	The error message(s) that the Analyzer detected during analysis.
page	The page number of the listing on which the error message is printed.

ERROR MESSAGES

The Analyzer can detect two kinds of errors:

- Operational Errors

  Errors that occur when you invoke the Analyzer or errors that occur during file operations. These errors are described in Chapter 3.

- Dump File Errors

  Errors within the dump file.

This section lists the Dump File Errors error messages and their descriptions. Dump file errors are errors that the Analyzer detects within the dump file. The Analyzer prints these errors in the section of the listing in which they occur. It also prints the error messages along with the page numbers on which they occur in the section of the listing entitled "Summary of Errors Detected by Analysis."

Message	Description
Nucleus entry or data segment corrupted, analysis terminated	The Analyzer uses the Nucleus interrupt vector to locate the Nucleus code segment. It then uses the code segment to find the data segment. The Analyzer uses the data segment as the basis for analysis. If any item in this chain is damaged, the Analyzer cannot function correctly.
Internal iRMX 86 <type> field corrupted at <BBBB>:<0000>	The Analyzer discovered an error in an internal operating system structure of <type> BYTE, WORD, or POINTER. The problem is located at base <BBBB> and offset <0000>.
Internal iRMX 86 <type> field corrupted at <BBBB>:<0000> Object token:<cccc>	The Analyzer discovered an error in an internal operating system structure of <type> BYTE, WORD, or POINTER. The problem is located at base <BBBB> and offset <0000>. The error is in the object whose token is <cccc>.
Stack overflow	This message appears when a task overflows its stack segment.
Registers not available	The processor's registers were not available to the Dumper. This message appears only under the "Current Processor State" portion of the listing.

Task stack segment not distinct: above display may contain other data in addition to stack	The task's stack is in a segment that was not allocated when the task was created. This message appears following the task segment listing because the Analyzer cannot distinguish stack data from other data in the segment. Therefore, this particular message indicates a lack of information necessary to the Crash Analyzer rather than a problem.
Task SS:SP not known: stack segment not displayed	The Analyzer could not find a valid stack segment:stack pointer (SS:SP). This message appears following a task segment display. This particular message indicates a lack of information necessary to the Crash Analyzer rather than a problem.
Unable to locate complete <block name>. Missing area is <address> for <size> bytes	The Analyzer could not find the entire <block name>. The block name is one of the following: connection object job directory task stack NPX save area composite list mailbox cache logical device object segment contents The <address> is the base and offset address of the missing information. The <size> is the size of the missing information in hexadecimal.
Unable to located job pool information	This message appears in the pool report for a job when the Analyzer cannot find the information describing a job's pool.
Unable to locate list of <object type> in job <job token>	The Analyzer cannot find the list of a particular object type for a job. This message appears in place of the list of an object type in the section entitled, "Objects Contained by Job" The <object type> is one of the following: child jobs tasks mailboxes semaphores regions segments extensions composites

	The <job token> is the token for the job in which the Analyzer cannot find the object type.
Unable to locate file node for <connection token>	This message appears when the Analyzer is unable to read the contents of the fnode because the pointer to the fnode has been destroyed.
Unable to locate device descriptor for connection <connection token>	This message appears when the Analyzer cannot find the device descriptor for a connection. This may happen because one of your tasks wrote over and internal data structure.
Unable to locate object queued on mailbox <token>. Token of missing object is <object token>	This message appears in the "Mailbox Report" when the Analyzer cannot find an object queued at the mailbox.
End of stack segment not known; stack segment not displayed	This message appears in the "Task Report" when the Analyzer cannot find the end of the stack segment.

LINK ERROR

The iRMX 86 Operating System maintains tokens in doubly-linked lists. So, whenever a listing contains a token, the Analyzer automatically checks the validity of that token by looking at the token's forward links.

A forward link error means that the iRMX 86 data structures have been damaged or destroyed. The most common reason for this problem is overwriting. You or one of your tasks may have accidentally written over part of the Operating System's data structures and/or code. Another possible reason for the problem (if you are using a non-maskable interrupt) could be that you interrupted the Nucleus while it was setting up the links.

If a token's forward link is bad, the Analyzer generates a forward link error message along with the information that the particular listing usually displays. The forward link error message is as follows:

```
    Forward link ERROR:  <aaaa> --> <bbbb>  ?<cccc> <-- <bbbb>
```

The arrows represent links. A right pointing arrow represents a forward link. The object with the token <aaaa> is linked forward to the object with token <bbbb>. The object with the token <bbbb> should be linked back to the object with the token <aaaa> rather than <cccc>. Therefore, the Crash Analyzer assumes the link from <aaaa> to <bbbb> is incorrect and terminates the analysis of the objects in the portion of the listing in which the error appears.

***

INDEX

Primary references are <u>underscored</u>.

***

iRMX™ 86 BOOTSTRAP LOADER REFERENCE MANUAL

CONTENTS

PAGE

***

CHAPTER 1 INTRODUCTION

The Bootstrap Loader is a means of loading part or all of an application from secondary storage into RAM, either upon system reset or under program control. With the Bootstrap Loader you have flexibility that can simplify system maintenance and increase the versatility of your hardware.

The Bootstrap Loader operates in two stages. The first stage determines the location of the second stage and the name of the load file, then loads part of the second stage and passes control to the second stage. The second stage finishes loading itself, transfers the load file into memory, and passes control to the load file. The load file usually consists of an iRMX 86 application system. Getting the load file into memory and passing control to it is the objective of the bootstrap loading process.

A device driver is a small program that interfaces between your software and a hardware device (or a controller for the device). When you perform Bootstrap Loader configuration, which is independent of application system configuration with the ICU, you specify the device drivers that the Bootstrap Loader requires. As you complete the Bootstrap Loader configuration process, the device drivers needed for bootstrap loading -- which are distinct from (although possibly identical to) the drivers needed by the application itself -- are linked in automatically.

The following sections contain miscellaneous facts that will enable you to understand the later discussion about incorporating the Bootstrap Loader into your system.

THE FIRST STAGE OF THE BOOTSTRAP LOADER

The first stage consists of two parts. One part is the code for the first stage. It varies from 100 to 1000 bytes (and averages about 500 bytes) in length, depending upon the options you request during configuration. The other part is a set of minimal device drivers the first and second stages need to accomplish their objectives.

When the bootstrap loading process begins, the first stage can be in either of two places. If you are still developing your application, you can keep your first stage in secondary storage on your development system, then load it and start it running by means of the iSBC 957B package or the iSDM 86 or iSDM 286 System Debug Monitor. You can also burn the first stage into ROM along with the iSDM 86 or iSDM 286 monitor. When your application is finished and ready to use, you will probably burn the first stage into ROM, so it can be invoked when you turn on or reset your system.

THE SECOND STAGE OF THE BOOTSTRAP LOADER

Unlike the first stage, the second stage is not configurable. That is, it is always the same -- its size is less than 8K bytes -- and does not depend on the application in any way. Because of this, the software that formats iRMX 86-based random access volumes places the random-access version of the second stage on every named volume it formats, so it is always available for loading applications residing on random-access devices.

When the application system begins to run, RAM that had been used or occupied by the first and/or second stage becomes part of the memory pool for the application system.

NOTE

You must ensure that the memory locations occupied by the first stage, the second stage, and the load file (application system) are mutually non-overlapping.

THE LOAD FILE

The load file must be placed on an iRMX 86-based named volume of secondary storage. Recall that this volume also contains the second stage of the Bootstrap Loader.

DEVICE DRIVERS

For every bootstrap device in your system, you must include a device driver. As part of the iRMX 86 product, Intel has provided you with many device drivers that are specifically for bootstrap loading. These drivers are for the following controllers:

- iSBC 204 Flexible Diskette Controller
- iSBC 206 Disk Controller
- iSBC 208 Flexible Disk Drive Controller
- iSBC 215 Winchester Disk Controller
- iSBX 218A Flexible Disk Controller when used with the iSBC 215 controller
- iSBX 218A Flexible Disk Controller when used on a CPU board
- iSBC 220 SMD Disk Controller

- iSBX 251 Bubble Memory Controller
- iSBC 254 Bubble Memory Controller
- SASI (Shugart Associates Systems Interface) Peripheral Bus Controller
- SCSI (Small Computer Systems Interface) Peripheral Bus Controller

These drivers are small, ranging from 300 to 1000 bytes in length, and averaging about 500 bytes.

If you need additional device drivers, see Chapter 4.

***

CHAPTER 2 CONFIGURATION

The key to using the Bootstrap Loader is to ensure that the first stage is properly configured into your application. How to do that is the subject of this chapter. (Recall that the second stage is constant and therefore does not have to be configured.)

Configuring the first stage of the Bootstrap Loader consists of editing three or more configuration files. If you have devices for which Intel does not supply a device driver, you must prepare a device driver for each of them. Chapter 4 describes how to do this.

The configuration files are the following:

BS1.A86	This assembly language source file consists primarily of macros that describe the device units that can be used for bootstrap loading and the manner in which the bootstrap device and load file are to be selected.
BSERR.A86	This assembly language source file consists primarily of macros that tell the Bootstrap Loader what to do when bootstrap loading is not successful.
B204.A86 B206.A86 B208.A86 B215.A86 B218.A86 B251.A86 B254.A86 BSASI.A86 BSCSI.A86	These assembly language source files contain configuration information about device drivers that your bootstrap system can use.
BS1.CSD	This SUBMIT file contains the commands needed to assemble the preceding source files, to link together the resulting modules (and any others that you supply), and to locate the resulting object module.

The files requiring editing are BS1.A86, BSERR.A86, and BS1.CSD.

BS1.A86 CONFIGURATION FILE

The BS1.A86 file, shown in Figure 2-1, consists of two INCLUDE statements and several macros. The BS1.INC file contains the definitions of the macros in the BS1.A86 file.

```
 name bsl

$include(:fl:bcico.inc)
$include(:fl:bsl.inc)

%cpu(8086)
;iAPX_186_INIT(y,ofc38h,none,80bbh,none,003bh)  ; 188/48 board
;iAPX_186_INIT(y,none,none,80bbh,none,0038h)    ; 186/03 and 186/51
                                                ;     boards
%console
%manual
%auto
%loadfile
%defaultfile('/system/rmx86')
%retries(5)

;cico

;    iSBC 86/05/12A/14/30
;serial_channel(8251a,0d8h,2,8253,0d0h,2,2,8)
;
;    iSBC 351 (on iSBX #0)
;serial_channel(8251a,0A0h,2,8253,0B0h,2,2,8)
;
;    8MHz iSBC 186/03/51
;serial_channel(8274,0d8h,2,80186,0ff00h,2,0,0dh)
;serial_channel(8274,0dah,2,80186,0ff00h,2,1,0dh)
;serial_channel(8274,0dah,2,80130,0e0h,2,2,034h)
;
;    6MHz iSBC 186/03/51
;serial_channel(8274,0d8h,2,80186,0ff00h,2,0,0ah)
;serial_channel(8274,0dah,2,80186,0ff00h,2,1,0ah)
;serial_channel(8274,0dah,2,80130,0e0h,2,2,027h)
;
;    iSBC 188/48 SCC #1
;serial_channel(82530,0d0h,1,82530,0d0h,1,0,0eh,a)
;serial_channel(82530,0d2h,1,82530,0d2h,1,0,0eh,b)
;
;
;
;
;
%device(f0, 0, deviceinit204, deviceread204)
%device(f1, 1, deviceinit204, deviceread204)
%device(f2, 2, deviceinit204, deviceread204)
%device(f3, 3, deviceinit204, deviceread204)
%device(af0, 0, deviceinit208gen, deviceread208gen)
%device(af1, 1, deviceinit208gen, deviceread208gen)
%device(af2, 2, deviceinit208gen, deviceread208gen)
%device(af3, 3, deviceinit208gen, deviceread208gen)
```

Figure 2-1. First Stage Configuration File BS1.A86

```
    %device(d0, 0, deviceinit206, deviceread206)
    %device(w0, 0, deviceinit215gen, deviceread215gen)
    %device(wf0, 8, deviceinit215gen, deviceread215gen)
    %device(wf1, 9, deviceinit215gen, deviceread215gen)
    %device(wf2, 10, deviceinit215gen, deviceread215gen)
    %device(wf3, 11, deviceinit215gen, deviceread215gen)
    %device(pmf0, 0, deviceinit218Agen, deviceread218Agen)
    %device(bx0, 0, deviceinit251, deviceread251)
    %device(b0, 0, deviceinit254, deviceread254)
    %device(sa0, 0, deviceinitsasi, devicereadsasi)
    %device(sc0, 0, deviceinitscsi, devicereadscsi)
    %end
```

Figure 2-1. First Stage Configuration File BS1.A86 (continued)

The following sections describe the functions of the macros in the BS1.A86 file. For each macro, if a percent sign (%) precedes the name, then the macro is included (invoked). If a semicolon (;) precedes the name, then the macro is treated as a comment and is not included.

The BS1.A86 file does not specifically mention iSBC 220 SMD devices because they are covered by the entries containing "215".

In each %DEVICE macro shown in Figure 2-1 as having "gen" as a suffix on its last two parameters, those parameters can also be present without the suffix. That is, for each of those macros, Intel has supplied two versions of the device$init and device$read procedures, one with the "gen" suffix and one without the suffix. The "gen" (for general) versions, which provide automatic device recognition (see Appendix A), require more (about 500 bytes) code.

%CPU MACRO

You must include the %CPU macro, to identify the type of CPU that performs the bootstrap loading operation.

The form of the %CPU macro is:

```
%CPU(cpu_type)
```

where:

cpu_type	8086, 8088, 80186, 80188, or 80286. These are informal names for the Intel processors whose formal names are iAPX 86, iAPX 88, iAPX 186, iAPX 188, and iAPX 286, respectively.

iAPX_186_INIT MACRO

The iAPX_186_INIT macro specifies the initial chip select and mode values for 80186 and 80188 CPUs. Include this macro if and only if the CPU type is either 80186 or 80188.

The form of the %iAPX_186_INIT macro is:

```
    %iAPX_186_INIT(RMX, UMCS, LMCS, MMCS, MPCS, PACS)
```

where RMX must contain "y" as it does in the file. The remaining parameters define initial values for the chip-select control registers. They stand, respectively, for upper-memory chip-select, lower-memory chip-select, midrange-memory chip-select, memory-peripheral chip-select, and peripheral-address chip-select block address. These registers are described in the data sheets for the iAPX 186 and iAPX 188 processors.

%CONSOLE, %MANUAL, AND %AUTO MACROS

The %CONSOLE, %MANUAL, and %AUTO macros let you specify how the first stage is to identify the load file and the device where the file will be found.

You can include any combination of the %CONSOLE, %MANUAL, and %AUTO macros. Because including %MANUAL causes the automatic inclusion of both %CONSOLE and %AUTO, there are five functionally-distinct combinations of these macros. The following indicates the significance of each of the five combinations.

None

(Requires that the device list, defined by means of the %DEVICE macro, have only one entry.)

- It (the Bootstrap Loader) tries once to load from the device in the device list.
- It tries once to load the file with the default pathname (either the system default or one you define by means of the optional %DEFAULTFILE macro).

%CONSOLE only

(Requires that the device list have only one entry.)

- It tries once to load from the device in the device list.
- It issues an asterisk (*) prompt for a pathname at the application system terminal and then tries once to load the file the operator specifies.
    - If a pathname is entered, it loads the file with that pathname.
    - If only <cr> is entered, loads the file with the default pathname.

%MANUAL only

(Requires a device list with at least one entry.)

- It issues an asterisk (*) prompt for a pathname at the application system terminal.
- It chooses a device depending upon the operator's response.
  - If a device name is entered, it loads from the device with that device name. It tries to load until the device becomes ready or until no more tries are allowed (as limited by the optional %RETRIES macro).
  - If no device name is entered before the carriage return, it looks for a ready device by searching through the list of devices (in the order in which they appear in the BS1.A86 file). The search continues until a ready device is found or until no more tries are allowed (as limited by the optional RETRIES macro). If it finds a ready device, it loads from that device.
- It chooses a file depending upon the operator's response to the prompt.
  - If a pathname is entered, it tries once to load the file with that pathname.
  - If no file name is entered, it tries once to load the file with the default pathname.

%AUTO only

(Requires a device list with at least one entry.)

- It looks for a ready device by searching through the list of devices (in the order in which they appear in the BS1.A86 file). The search continues until a ready device is found or until no more tries are allowed (as limited by the optional RETRIES macro).
- If it finds a ready device, it tries once to load the file with the default file name.

%AUTO and %CONSOLE

(Requires a device list with at least one entry.)

- It issues an asterisk (*) prompt for a pathname at the application system terminal.
- If the operator responds with a pathname that contains no device name, it looks for a ready device by searching through the list of devices (in the order in which they appear in the BS1.A86 file). The search continues until a ready device is found or until no more tries are allowed (as limited by the optional %RETRIES macro).

- If it finds a ready device or the operator responds with a pathname containing a device name, it tries once to load the file indicated by the operator's response.
  - If a pathname is entered, it tries to load the file with that pathname.
  - If only <cr> is entered, it tries to load the file with the default pathname.

In the foregoing macro descriptions, there are several references to an asterisk (*) prompt. If you have a monitor in PROM, with a pointer to its location in position 3 of the interrupt vector table, then when responding to this prompt you can use the Bootstrap Loader's Debug switch, which is described in Chapter 3.

The forms of the %CONSOLE, %MANUAL, and %AUTO macros are:

```
%CONSOLE

%MANUAL

%AUTO
```

%LOADFILE MACRO

The %LOADFILE macro causes the Bootstrap Loader to display at the console the pathname of the file it loads. It displays the pathname after loading the second stage and before loading the load file. The form of the %LOADFILE macro is:

```
%LOADFILE
```

%DEFAULTFILE MACRO

The %DEFAULTFILE macro specifies the hierarchical path of the default file. Its form is:

```
%DEFAULTFILE(pathname)
```

where pathname is the hierarchical path of the file enclosed in single quotes, as, for example, '/SYSTEM/TEST/RMX86'. If this macro is omitted, the pathname '/SYSTEM/RMX86' is assumed.

Do not omit this macro if you include the %LOADFILE macro.

%RETRIES MACRO

The %RETRIES macro, when included along with the %AUTO or %MANUAL macro, limits the number of times that the first stage goes through the device list in search of a ready device. If this macro is not included along with %AUTO or %MANUAL, and no device in the list is ready, then the search continues indefinitely. The form of the %RETRIES macro is:

```
    %RETRIES(number)
```

where number, which must lie in the range 1 through OFFFEH, is the maximum number of times the first stage checks each device for a ready condition.

%CICO MACRO

The %CICO macro specifies that console input and output are to be performed by standalone CI and CO routines, that is, routines that are not part of an iSDM 86, iSDM 286, or iSBC 957B monitor. If you include the %CICO macro, you must do some other things as well, depending upon whether the CI and CO routines you want to use are your own or those supplied by Intel.

If you use the Intel-supplied standalone CI and CO routines, you must do the following:

- Change the line in the BS1.CSD file that reads

  ```
  &    :fl:bcico.obj, &
  ```

  to

  ```
       :fl:bcico.obj, &
  ```

- Include exactly one instance of the %SERIAL_CHANNEL macro (described next) in the BS1.A86 file.

If you supply your own standalone CI and CO routines, you must do the following:

- Change the line in the BS1.CSD file that reads

  ```
  &    :fl:bcico.obj, &
  ```

  to

  ```
       :fl:mycico.obj, &
  ```

  where mycico.obj is an object file containing the CI and CO routines and a file called CINIT, which performs initialization functions required to prepare the console for input and output operations.

- Include no instances of the %SERIAL_CHANNEL macro.

The form of the %CICO macro is:

```
    %CICO
```

%SERIAL_CHANNEL MACRO

Your CPU board can communicate over a serial channel by means of either an 8251A USART, an 8274 Multi-Protocol Serial Controller, or an 82530 Serial Communications Controller. The %SERIAL_CHANNEL macro, which requires you to include the %CICO macro, allows you to specify which serial controller device your CPU board uses as well as information that defines the use characteristics of the device.

You can omit this macro if your system does not use a terminal during bootstrap loading, if your supply your own CI and CO routines, or if you system use the iSDM 86, iSDM 286, or iSBC 957B monitor. Otherwise, include one instance of it in your BS1.A86 file for the serial controller device that supports the terminal your system uses for bootstrap loading. Including multiple %SERIAL_CHANNEL macros causes an assembly error when the BS1.CSD file runs.

The format of the %SERIAL_CHANNEL macro is as follows:

```
    %SERIAL_CHANNEL (serial_type, serial_base_port, serial_port_delta,
                     counter_type, counter_base_port, counter_port_delta,
                     baud_counter, count, flags)
```

where:

serial_type	The serial controller device you are using. The valid values are 8251A, 8274, and 82530.
serial_base_port	The 16-bit address of the base port used by the device. This port varies according to the type of the device and, if applicable, the channel used on the device, as follows:

8251A	Data Register Port
8274 Channel A	Channel A Data Register Port
8274 Channel B	Channel B Data Register Port
82530 Channel A	Channel A Command Register Port
82530 Channel B	Channel B Command Register Port

serial_port_delta	The number of bytes between consecutive ports used by the serial device.
counter_type	The type of device containing the timer your CPU board uses to generate a baud rate for the serial device defined by this macro. The valid values are 8253, 8254, 80130, 80186, 82530, and NONE. Specifying NONE implies that the baud rate timer is automatically initialized and the Bootstrap Loader does not have to perform this function.

```
;
; *-*-*   BS1.CSD   *-*-*
;
;   Generate the iAPX 86, 88 Bootstrap Loader V5.0 first stage.
;
;   Invocation:  submit bsl(first stage location, second stage location)
;
run
;
asm86 :f1:bs1.a86   macro(50) object(:f1:bs1.obj)   print(:f1:bs1.lst)
asm86 :f1:bserr.a86 macro(50) object(:f1:bserr.obj) print(:f1:bserr.lst)
asm86 :f1:b204.a86  macro(50) object(:f1:b204.obj)  print(:f1:b204.lst)
asm86 :f1:b206.a86  macro(50) object(:f1:b206.obj)  print(:f1:b206.lst)
asm86 :f1:b208.a86  macro(50) object(:f1:b208.obj)  print(:f1:b208.lst)
asm86 :f1:b215.a86  macro(50) object(:f1:b215.obj)  print(:f1:b215.lst)
asm86 :f1:b218a.a86 macro(50) object(:f1:b218.obj)  print(:f1:b218.lst)
asm86 :f1:b251.a86  macro(50) object(:f1:b251.obj)  print(:f1:b251.lst)
asm86 :f1:b254.a86  macro(50) object(:f1:b254.obj)  print(:f1:b254.lst)
asm86 :f1:bsasi.a86 macro(50) object(:f1:bsasi.obj) print(:f1:bsasi.lst)
asm86 :f1:bscsi.a86 macro(50) object(:f1:bscsi.obj) print(:f1:bscsi.lst)
;

link86
    :f1:bs1.obj,                       &
    :f1:bserr.obj,                     &
&   :f1:bcico.obj,                     & ;for standalone serial channel
                                       & ;support
    :f1:b204.obj,                      &
    :f1:b206.obj,                      &
    :f1:b208.obj,                      &
    :f1:b215.obj,                      &
    :f1:b218.obj,                      &
    :f1:b251.obj,                      &
    :f1:b254.obj,                      &
    :f1:bsasi.obj,                     &
    :f1:bscsi.obj,                     &
    :f1:bs1.lib                        &
    to :f1:bs1.lnk print(:f1:bs1.mp1)  &
    nopublics except(firststage,boot_186,bootstrap_entry)
;
loc86 :f1:bs1.lnk                                      &
    addresses(classes(code(%0),stack(%1)))             &
    order(classes(code,code_error,stack,data,boot))    &
    noinitcode                                         &
    start(firststage)                                  &
 &  ; change above line to start(boot_186) if iAPX_186_INIT is invoked  &
    segsize(boot(1800H))                &
    map print(:f1:bs1.mp2)              &
    ; Add "bootstrap" to loc86 when locating the first stage in ROM
```

Figure 2-2. First Stage Configuration File BS1.CSD

```
;
exit
;
;   Bootstrap Loader first stage generation complete.
;
```

Figure 2-2. First Stage Configuration File BS1.CSD (continued)

counter_base_port	The 16-bit address of the base port used by the baud rate timer. This port varies according to the type of the device and, if applicable, the channel used on the device, as follows:

8253	Counter 0 Count Register Port
8254	Counter 0 Count Register Port
80130	ICW1 Register Port
80186	Use 0FF00H on all Intel boards
82530 Channel A	Channel A Command Register Port
82530 Channel B	Channel B Command Register Port

counter_port_delta	The number of bytes between consecutive ports used by the timer.
baud_counter	The baud rate-generating counter on the timer. The devices and the counters you can specify for them are as follows:

8253	0, 1, and 2
8254	0, 1, and 2
80130	2
80186	0, 1
82530	0

count	A value that, when loaded into the timer register, generates the desired baud rate. The method of calculating this value is described in the paragraphs following these parameter definitions.
flags	A value that, when present, specifies which channel of an 82530 Serial Communications Controller will serve as your serial controller. If you give any value except 82530 for the serial_type parameter, omit this parameter; that is, write the macro as if the count parameter is the last parameter. If you give 82530 as the value of the serial_type parameter, specify A (for Channel A) or B (for Channel B) for this parameter.

To derive the correct value for the count parameter, you must perform a short series of computations. The starting values for these computations are the desired baud rate and the clock input frequency to the timer.

The first computation yields a temporary value and depends upon the timer used, as follows:

```
temporary_value = (clock frequency in Hertz)/(baud rate x 16)

     if the timer is an 8253, 8254, 80130, or 80186, but

temporary_value = ((clock frequency in Hertz)/(baud rate x 2)) - 2

     if the timer is an 82530.
```

The second computation yields the fractional part of the temporary value, as follows:

```
fraction = temporary_value - INT (temporary_value)
```

where the INT function gives the integer portion of temporary_value.

The third and fourth computations yield the desired count value and another value, called error_fraction. The error_fraction value is then used to determine whether the calculated count value is feasible, given the clock frequency specified in the first computation. These computations, which are performed according to the size of the value of "fraction" from the second computation, are as follows:

```
count = INT (result) + 1
error_fraction = 1 - fraction

     if the value of "fraction" is greater than or equal to .5, but

count = INT (result)
error_fraction = fraction

     if the value of "fraction" is less than .5.
```

The fifth and final computation, which yields the percentage of error that occurs when the given clock frequency is used to generate the given baud rate, is as follows:

```
% error = (error_fraction / count) x 100
```

If the % error value is less than 3, then the calculated count value is appropriate and will lead to the desired baud rate being generated by the specified clock frequency. However, if the % error value is 3 or greater, you must do either or both of the following two things:

- Provide a higher clock frequency
- Select a lower baud rate

After choosing one or both of these options, go through the series of computations again so as to get a new value of "count" and to see whether the revised value of "% error" is less than 3.

Continue this process -- raise the clock frequency and/or lower the baud rate, then do the computations -- until you finally get a "% error" value lower than 3.

The % SERIAL_CHANNEL macro can generate the following error messages:

```
ERROR - invalid port delta for the Serial Device
ERROR - <ser_type> is an invalid Serial Port type
ERROR - Invalid port delta for the Baud Rate Timer
ERROR - 8253/4 Baud Rate Counter is not 0, 1, or 2
ERROR - 2 is the only valid 80130 Baud Rate Timer
ERROR - 80186 counter counter_type is not a valid baud rate counter
ERROR - <counter_type> is an invalid Baud Rate Timer type
ERROR - Counter 0 is the only valid 82530 baud rate counter
ERROR - 82530 channel must be specified as A or B only
ERROR - Baud Rate Count must be greater than 1
```

%DEVICE MACRO

The %DEVICE macro defines a device unit from which your application system can be bootstrap loaded. If the BS1.A86 file contains multiple %DEVICE macros, their order in the file is the order in which the first stage searches for a ready device unit. Recall that multiple %DEVICE macros may be included only if the %AUTO or %MANUAL macro is included. (Otherwise, there is an assembly error when the BS1.CSD file runs.) The form of the %DEVICE macro is:

```
%DEVICE(name, unit, device$init, device$read)
```

where:

name	The physical name of the device, not enclosed in quotes or between colons. The first stage passes the physical name to the second stage, which, in turn, passes it to the load file. If the Automatic Boot Device Recognition (see Appendix A) capability is configured into the load file, then the physical names in the %DEVICE macro invocations must match the device unit names in the load file. Otherwise, the load file will not initialize properly and could "hang."
unit	The number of this unit on this device.

device$init	The name of the device$init procedure of the device driver the first stage will call for this device unit. If you are using an Intel-supplied driver, specify the procedure name as shown in Figure 2-1. (You may omit the "gen" suffix; see the discussion of this topic earlier in this chapter.) If you are supplying your own driver, which you have written in accordance with the instructions in Chapter 4, use the name of the initialization procedure.
device$read	The name of the device$read procedure of the device driver the first stage will call for this device unit. If you are using an Intel-supplied driver, specify the procedure name as shown in Figure 2-1. (You may omit the "gen" suffix; see the discussion of this topic earlier in this chapter.) If you are supplying your own driver, which you have written in accordance with the instructions in Chapter 4, use the name of the read procedure.

%END MACRO

The %END macro is required at the end of the BS1.A86 and BSERR.A86 assembly language source files. Its form is:

```
%END
```

BSERR.A86 CONFIGURATION FILE

The BSERR.A86 file, shown in Figure 2-3, defines what the first stage of the Bootstrap Loader does if it cannot load the load file.

```
 name bserr

$include(:f1:bserr.inc)

;console
%text
%list

%again
;int3
;halt

%end
```

Figure 2-3. First Stage Configuration File BSERR.A86

The BSERR.A86 file consists of an INCLUDE statement and several macros. The BSERR.INC file in the INCLUDE statement contains the definitions of the macros in the BSERR.A86 file.

The following sections describe the functions of the macros in the BSERR.A86 file. For each macro, if a percent sign (%) precedes the name, then the macro is included (invoked). If a semicolon (;) precedes the name, then the macro is treated as a comment and is not included.

The first three macros, %CONSOLE, %TEXT, and %LIST, determine what the Bootstrap Loader displays at the console whenever a bootstrap loading error occurs. The other three macros, %AGAIN, %INT3, and %HALT, determine what recovery steps, if any, the Bootstrap Loader takes whenever a bootstrap loading error occurs. Only one of the latter three macros can be included in the BSERR.A86 file.

%CONSOLE MACRO

The %CONSOLE macro causes the Bootstrap Loader to display a brief message at the console whenever a bootstrap loading error occurs. This message indicates the nature of the error. The messages are given in Chapter 3. The form of the %CONSOLE macro is:

```
    %CONSOLE
```

This %CONSOLE macro is completely unrelated to the %CONSOLE macro in the BS1.A86 file. Be careful not to confuse them with each other.

%TEXT MACRO

The %TEXT macro resembles the %CONSOLE macro in that it causes the Bootstrap Loader to display a message at the console whenever a bootstrap loading error occurs. The advantage of the %TEXT macro is that its messages are longer and more descriptive. The disadvantage of the %TEXT macro is that it generates more code and therefore makes the assembled BSERR.OBJ file larger. The %TEXT macro has the form:

```
    %TEXT
```

If you include the %TEXT macro, the %CONSOLE macro is automatically included, as well.

%LIST MACRO

The %LIST macro causes the Bootstrap Loader to display a list of the ready device units whenever the operator enters an invalid device unit name. You may include this macro only if you include the %MANUAL macro in the BS1.A86 file, described earlier in this chapter. The %LIST macro has the form:

```
    %LIST
```

If you include the %LIST macro, the %CONSOLE and %TEXT macros are automatically included, as well.

%AGAIN MACRO

The %AGAIN macro causes the bootstrap loading sequence to return to the beginning of the first stage whenever a bootstrap loading error occurs. It is a good idea to include this macro if you include the %CONSOLE macro in the BSERR.A86 file, either directly or by including the %TEXT or %LIST macro. The form of the %AGAIN macro is:

```
    %AGAIN
```

Exactly one of the %AGAIN, %INT3, and %HALT macros must be included, or there will be an assembly error when the BS1.CSD file runs.

%INT3 MACRO

The %INT3 macro causes the Bootstrap Loader to execute an INT 3 (software interrupt) instruction whenever a bootstrap loading error occurs. If you are using the iSDM 86 or iSDM 286 System Debug Monitor or the iSBC 957B package, then the INT 3 instruction passes control to the monitor. Otherwise, the INT 3 instruction will not produce the desired results unless you have placed the appropriate address in position 3 of the interrupt vector table. The form of the %INT3 macro is:

```
    %INT3
```

Exactly one of the %AGAIN, %INT3, and %HALT macros must be included, or there will be an assembly error when the BS1.CSD file runs.

The %INT3 macro and the %HALT macro (described next) are reasonable choices if none of the %CONSOLE, %TEXT, and %LIST macros are included in the BSERR.A86 file.

%HALT MACRO

The %HALT macro causes the Bootstrap Loader to execute a halt instruction whenever a bootstrap loading error occurs. The form of the %HALT macro is:

```
%HALT
```

Exactly one of the %AGAIN, %INT3, and %HALT macros must be included, or there will be an assembly error when the BS1.CSD file runs.

The %HALT macro and the %INT3 macro are reasonable choices if none of the %CONSOLE, %TEXT, and %LIST macros are included in the BSERR.A86 file.

INTEL-SUPPLIED DEVICE DRIVER CONFIGURATION FILES

There is a separate configuration file for each device driver provided with the Bootstrap Loader. These files are named B204.A86, B206.A86, B208.A86, B215.A86, B218.A86, B251.A86, B254.A86, BSASI.A86, and BSCSI.A86. Each consists of an include statement and a macro call. The include statement always has the form:

```
$include(:f1:bxxx.inc)
```

where:

xxx	Either 204, 206, 208, 215, 218, 251, 254, SASI, or SCSI, depending upon the device driver.

The macro call has a form that depends upon the device driver. This form is discussed in the following sections. The default parameter values for the macros in these sections are compatible with the default parameter values of the iRMX 86 Interactive Configuration Utility.

%B204 MACRO

The %B204 macro has the form:

```
%B204(io_base, sector_size, track_size)
```

where:

io_base	I/O port address selected (jumpered) on the iSBC 204 controller board.
sector_size	Sector size for the device, in bytes.
track_size	Track size for the device, in bytes.

The default form of this macro in the B204.A86 file is:

```
%B204(0A0H, 128, 26)
```

%B206 MACRO

The %B206 macro has the form:

```
%B206(io_base)
```

where:

io_base	I/O port address selected (jumpered) on the iSBC 206 controller board.

The default form of this macro in the B206.A86 file is:

```
%B206(068H)
```

%B208 MACRO

The %B208 macro has the form:

```
%B208(io_base)
```

where:

io_base	I/O port address selected (jumpered) on the iSBC 208 controller board.

The default form of this macro in the B208.A86 file is:

```
%B208(180H)
```

%B215 AND %B220 MACROS

The B215.A86 file contains two macros, of which you can use only one. They are the %B215 and the %B220 macros. Both of them have the form:

```
%Bxxx(wakeup, cylinders, fixed_heads, removable_heads, sectors,
     dev_gran, alternates)
```

where:

xxx	Either 215 or 220.
wakeup	Base address of the wakeup port

cylinders	Number of cylinders on the disk drive or drives. (Note that, if your %DEVICE macro for 215 or 220 devices in the BS1.A86 file has deviceunit215 (rather than deviceunit215gen) as its third parameter, then all iSBC 215 or iSBC 220 drives used by the Bootstrap Loader must have the same characteristics. That is, they must have the same number of cylinders per platter, fixed heads, removable heads, sectors per track, bytes per sector, and alternate cylinders. However, if the %DEVICE macro specifies deviceunit215gen, these restrictions do not apply and these values are not used.)
fixed_heads	Number of heads on fixed platters.
removable_heads	Number of heads on removable platters.
sectors	Number of sectors per track.
dev_gran	Number of bytes per sector.
alternates	Number of cylinders set aside as backups for cylinders having imperfections.

In the B215.A86 file, the default form of the %B215 macro is:

```
%B215(100H, 256, 2, 0, 9, 1024, 5)
```

and the default form of the %B220 macro is:

```
%B220(100H, 256, 2, 0, 9, 1024, 5)
```

%B218 MACRO

The %B218 macro has the form:

```
%B218(base_port_address, motor_flag)
```

where:

base_port-address	The base port address of this device unit, as selected on the iSBX 218A controller board.
motor_flag	A value indicating whether the motor of a 5 1/4" flexible diskette drive should be turned off after bootstrap loading. Specify Yes, which slows bootstrap loading, only if this device is not the system device. For Yes, specify OFFH, and for No, specify 0.

The default form of this macro in the B218.A86 file is:

```
%B218(80H, 00H)
```

%B251 MACRO

The %B251 macro has the form:

```
%B251(io_base, dev_gran)
```

where:

io_base	I/O port address selected (jumpered) on the iSBX 251 controller board.
dev_gran	Page size, in bytes.

The default form of this macro in the B251.A86 file is:

```
%B251(80H, 64)
```

%B254 MACRO

The %B254 macro has the form:

```
%B254(io_base, dev_gran, num_boards, board_size)
```

where:

io_base	I/O port address selected (jumpered) on the iSBC 254 controller board.
dev_gran	Page size, in bytes.
num_boards	Number of boards grouped in a single device unit.
board_size	Number of pages in one iSBC 254 board.

The default form of this macro in the B254.A86 file is:

```
%B254(0880H, 256, 8, 2048)
```

%BSASI MACRO

The %BSASI macro has the form:

```
%BSASI(a_port, b_port, c_port, control-port, init_command,
      init_byte_count, init_bytes)
```

where:

a_port	The address of Port A of the 8255 Programmable Peripheral Interface (PPI) used by this SASI driver.

b_port	The address of Port B of the 8255 PPI used by this SASI driver.
c_port	The address of Port C of the 8255 PPI used by this SASI driver.
control-port	The address of the control word register of the 8255 PPI used by this SASI driver.
init_command	The command that initializes the controller. (If the controller does not require initialization, use the SCSI driver instead of the SASI driver.) For the initialization command, look in the owner's manual for the controller. It might be labelled there either as a "command" or as an "opcode."
init_byte_ count	The number of initialization bytes that follow this parameter.
init_bytes	A list of initialization bytes, separated by commas, that define the characteristics of the drive. These values depend upon both the type of the controller and the type of the drive. The values can be found in the owner's manual for the controller.

NOTE

The BSASI macro is different than the other macros in that, if there are multiple occurrences of it in the BSASI.A86 file, then the corresponding devices must be either identical or completely compatible. That is, the devices must have identical specifications and can differ only in their unit number.

The default form of this macro in the BSASI.A86 file is:

```
    %BSASI(C8H, CAH, CCH, CEH, 0CH, 8, 01H, 32H, 6, 0, 0FFH, 0,
           0FFH, 0BH)
```

This default macro definition is for a Xebec S1410 5 1/4-inch Winchester disk controller and a Computer Memories, Inc. CMI-5419 19-megabyte Winchester disk drive.

%BSCSI MACRO

The %BSCSI macro has the form:

```
    %BSCSI(a_port, b_port, c_port, control_port, host_id, arbitrate)
```

where:

a_port	The address of Port A of the 8255 Programmable Peripheral Interface (PPI) used by this SCSI driver.
b_port	The address of Port B of the 8255 PPI used by this SCSI driver.
c_port	The address of Port C of the 8255 PPI used by this SCSI driver.
control_port	The address of the control word register of the 8255 PPI used by this SCSI driver.
host-id	The ID of the host computer on the SCSI bus.
arbitrate	A flag indicating whether bus arbitration is supported. Set to 0, which signifies that bus arbitration is not supported.

The SCSI driver can be used to bootstrap load from any Winchester device on the SCSI bus.

The default form of this macro in the BSCSI.A86 file is:

```
    %BSCSI(C8H, CAH, CCH, CEH, 80H, 00H)
```

The default macro definition is for an iSBC 186/03 board.

USER-SUPPLIED DRIVERS

If you want to bootstrap load your system from a device other than one controlled by an iSBC 204, 206, 208, 215, 218A, 220, 251, or 254 board, or one that interfaces with the SASI or SCSI driver, you must write your own init and read device driver procedures. In addition, you must specify their procedure names in the %DEVICE macro in the BS1.A86 file, and you must assemble them and link them to the rest of the Bootstrap Loader object files and libraries. Chapter 4 describes how to write the device driver procedures.

GENERATING THE BOOTSTRAP LOADER SYSTEM

To generate the bootstrap loading system, enter the command

```
    SUBMIT BS1.CSD(first_stage_address, second_stage_address)
```

where the parameters specify the low-memory addresses of the stages.

The size of the first stage area depends upon the device drivers in the first stage. If you use Intel-supplied drivers, the size is always less than 8K bytes, even with all of the drivers configured in at once.

Recall that the first stage can be either in PROM or in RAM.
The second stage area, which includes the code of the second stage and the data areas for both stages, consists of slightly less than 8K contiguous bytes. The second stage always resides in RAM.

***

CHAPTER 3
USING THE BOOTSTRAP LOADER

This chapter describes how to set up and invoke the Bootstrap Loader and what to do if it fails to perform as expected.

PREPARING TO USE THE BOOTSTRAP LOADER

There are four ways to bootstrap load your application. The key to each of these methods is the first stage of the Bootstrap Loader: where you put it and how you invoke it. The four methods are as follows:

- Place the first stage, configured for standalone operation, in PROM. In this case, bootstrap loading commences -- that is, the first stage begins to run -- when you turn on the system hardware or press the RESET button.

- Place the first stage in secondary storage, and then load it by means of external commands. Doing this requires you to use the iSDM 86, iSDM 286, or iSBC 957B monitor, or an ICE in-circuit emulator, first to load the first stage into RAM and then to invoke the first stage.

- Augment the iSDM 86, iSDM 286, or iSBC 957B monitor by reconfiguring the first stage of the Bootstrap Loader to include the device driver(s) needed for bootstrap loading, and program new PROMs with the combination of the monitor and the first stage of the Bootstrap Loader. With this method, you initiate bootstrap loading by means of the B command of the monitor.

- Place the first stage in secondary storage, and then load it programmatically.

In the first method, you must add the BOOTSTRAP control to the LOC86 command used in the BS1.CSD file, as indicated in the last comment in that file. Otherwise, each of the first two methods is straightforward and therefore is not described in this manual.

The instructions for using the third method lie outside of this chapter. To use this method with the iSDM 86 monitor, follow the instructions given in the iSDM 86 SYSTEM DEBUG MONITOR REFERENCE MANUAL. In addition, Appendix B of this manual has a short section that describes a required modification of a file that is listed in the iSDM 86 manual.

To use the third method with the iSDM 286 monitor, refer to Appendix B of this manual, as well as the iSDM 286 SYSTEM DEBUG MONITOR REFERENCE MANUAL.

To use the third method with the iSBC 957B monitor, follow the directions given in the USER'S GUIDE FOR THE iSBC 957B iAPX 86, 88 INTERFACE AND EXECUTION PACKAGE.

Note that error handling, as described later in this chapter, does not take place when you use the third method of bootstrap loading with the iSBC 957B package.

The rest of this section gives instructions for using the fourth method.

Although bootstrap loading is performed usually in response to an external event, it can be initiated by an executing program by means of a call to the PUBLIC symbol BOOTSTRAP_ENTRY. To prepare for such a call, do the following:

- Place a call to BOOTSTRAP_ENTRY in the code of the invoking program, and define BOOTSTRAP_ENTRY as an EXTERNAL symbol there. The form of the call is:

  ```
  CALL BOOTSTRAP_ENTRY(@filename)
  ```

  where:

filename	An ASCII string containing either the pathname of the load file followed by a carriage return, or only a carriage return. If the string contains only a carriage return, then the default file, as defined by the %DEFAULTFILE macro in the BS1.A86 configuration file, is loaded. Otherwise, the file whose pathname is contained in the string is loaded.

  The call must follow the PL/M-86 LARGE model of segmentation. (Even though this is a call, rather than a jump, it does not return.)

- Link the calling program to a version of the first stage of the Bootstrap Loader. You can do this by following the BS1.CSD file as a model, with the following changes:

  - Place the calling program in the link sequence.

  - If appropriate, "comment out" the locate sequence.

OPERATOR'S ROLE IN BOOTSTRAP LOADING

Depending upon the method used for Bootstrap Loading, an operator might be required to enter the name of the bootstrap device, the name of the load file, or both names. (Another possibility, depending upon how the Bootstrap Loader is configured, is that the operator can enter neither device name nor file name. This section refers to what the operator enters as the specification of the load file.) Along with the specification of the load file, the operator can specify, by means of the

Debug switch, that control should pass to the monitor after loading has been completed. These are the subjects of this section.

SPECIFYING THE LOAD FILE

There are two times at which an operator can enter the specification of a load file. One time is when one of the monitors has issued a period (.) prompt. In that case, the operator can enter the monitor's B (bootstrap) command, followed by the specification. The other time is when the first stage of the Bootstrap Loader has issued an asterisk (*) prompt at the terminal. When this prompt appears on the screen, the first stage waits for an operator to enter the specification of the load file.

Once the period or asterisk prompt has been issued, the specification that the operator enters depends three things. They are:

- Which file is the load file.
- Which device unit contains the load file.
- Which of the %CONSOLE, %MANUAL, and %AUTO macros were in the BS1.A86 file when the present configuration of the Bootstrap Loader was defined.

For a discussion of the possible operator actions and their effects, see the description of the %CONSOLE, %MANUAL, and %AUTO macros in Chapter 2.

THE DEBUG SWITCH

Along with the specification of the load file, the operator can include the Bootstrap Loader's Debug (D) switch. When specified, the Debug switch instructs the second stage of the Bootstrap Loader to do the following immediately after loading has been completed:

- Set a breakpoint at the first instruction to be executed by the system.
- Pass control to the monitor, which displays a "*BREAK* at xxxx:xxxx" (iSDM 86 and iSBC 957B monitors) or an "Interrupt 3 at <xxxx:xxxx>" (iSDM 286 monitor) message at the terminal, issues its prompt, and waits for a command from the terminal. (To start up the loaded system, enter "G<cr>".)

One advantage of the Debug switch is that the monitor's message tells you that the loading process is successful. When a system fails, it is sometimes difficult to determine whether the bootstrap loading was unsuccessful or whether the system loaded successfully and then failed during initialization. The presence or absence of the message makes this clear when you use the Debug switch.

The Debug switch also allows you to alter the contents of specific memory locations before your system begins to run.

To use the Debug switch with a monitor's period (.) prompt, follow the B command in the command line by the letter "D", which, in turn, can be followed by the pathname of the load file. For example, any of the following command lines invokes the Bootstrap Loader with the Debug switch:

```
.bd

.b d

.bd /system/rmx86

.b  d  :w0:system/rmx86
```

Similarly, the way to use the Debug switch with the first stage's asterisk (*) prompt is to precede the load file specification with the letter "D." Examples of this are:

```
*d

*  d

*d /system/rmx86

* d  :w0:system/rmx86
```

The only restriction concerning the use of spaces in these command lines is that there must be a space between the letter "D" and the pathname of the load file.

Note that the Debug switch is available only in second stages residing on secondary storage volumes that have been formatted using the Release 6 (or later) versions of the iRMX 86 Format command. If you use the Debug switch with older second stages, the letter "D" is ignored, and the loadfile is loaded and run without the effects the Debug switch has when used with Release 6-compatible volumes.

ANALYZING BOOTSTRAP LOADING FAILURES

The Bootstrap Loader has the ability to display messages on the screen when bootstrap loading is not successful. As you saw in Chapter 2, the %CONSOLE, %TEXT, and %LIST macros in the BSERR.A86 file determine whether such messages are to be displayed, how detailed the messages are, and under what circumstances they are to be displayed. This section describes two ways of analyzing bootstrap loading errors: first, when messages are being displayed; and second, when there are no messages.

After responding to an error by pushing a word onto the stack and optionally displaying a message, the Bootstrap Loader either tries again, passes control to a monitor, or halts, depending upon whether your BSERR.A86 file contains a %AGAIN, %INT3, or %HALT macro.

WITH DISPLAYED ERROR MESSAGES

If your BSERR.A86 file contains the %CONSOLE, %TEXT, or %LIST macro, then the Bootstrap Loader displays an error message at your terminal whenever a failure occurs in the bootstrap loading process. The message consists of one or two parts. The first part, which is always displayed, is a numerical error code. The second part, which is displayed only if the %TEXT or %LIST macro was included, is a short verbal description of the error.

Each numerical error code has two digits. The first digit indicates, if possible, the stage of the bootstrap loading process where the error occurred. The second digit distinguishes among types of errors that can occur in a particular stage. There are three possible values for the first digit:

First Digit	Stage
0	Can't tell
1	First
2	Second

The error codes, their abbreviated display messages, and their causes and meanings are as follows.

Error Code: 01
Description: I/O error

An I/O error occurred at some undetermined time during the bootstrap loading process, but it is not clear when the error occurred. To help you further diagnose this problem if the %CONSOLE macro is included, the Bootstrap Loader places a code in the high-order byte of the word it pushes onto the stack. This byte identifies the driver for the device that produced the error, as follows:

Code	Driver
04H	204
06H	206
08H	208
15H	215 (with or without 218A) or 220
18H	218A on CPU board
51H	251
54H	254
0E0H	SCSI
0E1H	SASI
other (in range A0H-DFH)	driver for your custom device

Note that this device code is overwritten during the printing of the description in case the %TEXT or %LIST macro has been included.

The last entry in the list of device codes assumes that you have written a device driver for your device and have identified the driver by some code in the indicated range -- other values are reserved for Intel drivers. For information about how to incorporate this code into the driver, see Chapter 4.

Error Code: 11
Description: Device not ready.

The specific device designated for bootstrap loading is not ready. This error occurs only when your BSERR.A86 file does not contain the %AUTO macro. Therefore, either the operator has specified a particular device or there is only one device in the Bootstrap Loader's device list, and the device is not ready.

Error Code: 12
Description: Device does not exist. (If BSERR.A86 contains the %LIST macro, the display then shows the list of known devices.)

The device name entered at the console does not have an entry in the Bootstrap Loader's device list. This error occurs only when your BSERR.A86 file contains the %MANUAL macro and you enter a device name, but the device name you entered is not known to the Bootstrap Loader. After displaying the message, the Bootstrap Loader displays the names of the devices in its device list.

Error Code: 13
Description: No device ready.

None of the devices in the Bootstrap Loader's device list are ready. This error occurs only when your BSERR.A86 file contains the %AUTO or %MANUAL macro and you do not enter a device name at the console.

Error Code: 21
Description: File not found.

The Bootstrap loader was not able to find the indicated file on the designated bootstrap device. This is the default file if no pathname was entered at the console. Otherwise, it is the file whose pathname was entered.

Error Code: 22
Description: Bad checksum.

While trying to load the load file, the Bootstrap Loader encountered a checksum error. Each file consists of several records, and associated with each record is a checksum value that specifies the numerical sum (ignoring overflows) of the bytes in the record. When the Bootstrap Loader loads a file, it computes a checksum value for each record and compares that value to the recorded checksum value. If there is a discrepancy for any record in the file, it usually means that one or more bytes of the file have been corrupted, so the Bootstrap Loader returns this message instead of continuing the loading process.

Error Code: 23
Description: Premature end of file.

The Bootstrap Loader did not find the required end-of-file records at the end of the load file.

Error Code:	24
Description:	No start address found in input file.

The Bootstrap Loader successfully loaded the load file but was unable to transfer control to the file because when it got to the end of the file it still had not found initial CS and IP values.

WITHOUT DISPLAYED ERROR MESSAGES

In most cases, by observing the behavior of the Bootstrap Loader when it fails to load the application successfully, you can determine the cause of the failure and take steps to correct it. Table 3-1 shows the correlation between the behavior of the Bootstrap Loader and most of the possible causes of its failure. The table assumes that the Bootstrap Loader is set up to halt if it detects an error. Before halting, the Bootstrap Loader places the error code into the CX register.

Another possible cause of failure of the Bootstrap Loader, the effects of which are completely unpredictable, is that the device controller block (as determined by the device's wake-up address) can be corrupted. To avoid this kind of failure, see that neither the second stage nor the load file overlaps the device controller block for the device.

Table 3-1. Postmortem Analysis Of Bootstrap Loader Failure

Behavior Of Loader	Possible Causes
Bootstrap loading fails in the first stage.	The indicated device is not ready or is not known to the Bootstrap Loader. An I/O error occurred during the first stage operation.
Bootstrap loading fails in the second stage.	The indicated file is not on the device. The file had no end-of-file record or no start address. The file contained a checksum error. An I/O error occurred during the second stage operation.
Bootstrap Loader enters second stage, but does not halt or pass control to the loaded file.	The Bootstrap Loader is attempting to load the system on top of the second stage. The Bootstrap Loader is attempting to load the system into nonexistent memory.

***

CHAPTER 4 WRITING A DRIVER FOR A BOOTSTRAP LOADING DEVICE

The iRMX 86 Bootstrap Loader can be configured to run with many kinds of devices. If you plan to use one of the devices for which Intel supplies a device driver, you may skip this chapter.

If you want to use the Bootstrap Loader with a device other than those supported by Intel, you must write your own device driver. The purpose of this section is to provide you with guidelines for writing a customized driver.

Two procedures must be included in every device driver for the Bootstrap Loader. The initialization procedure initializes the bootstrap device. The reading procedure loads information from the device into RAM.

The rest of this chapter refers to the two procedures as DEVICE$INIT, and DEVICE$READ. However, you can give them any names you want during configuration. You must specify each of their names in a %DEVICE macro in the BS1.A86 file.

Both device driver procedures must conform to the LARGE model of segmentation of the PL/M-86 programming language. This means that the procedures must be FAR (not NEAR) and all pointers must be 32 bits long.

You may write the procedures in assembly language, rather than in PL/M-86, but if you do, you must adhere to the interfacing and referencing conventions of the PL/M-86 LARGE model.

DEVICE$INIT PROCEDURE

The DEVICE$INIT procedure must present the following PL/M-86 interface to the Bootstrap Loader:

```
DEVICE$INIT: PROCEDURE (unit) WORD PUBLIC;
          DECLARE unit     WORD;
        .
        . (code)
        .
END DEVICE$INIT;
```

where:

unit	The device's unit number, as defined during Bootstrap Loader configuration.

The WORD value returned by the procedure must be the device granularity, in bytes, if the device is ready, or zero if the device is not ready.

The following outline shows the steps that the DEVICE$INIT procedure must perform to be compatible with the Bootstrap Loader:

1. Test to see if the device is present. If it is not, return the value zero.

2. Initialize the device for reading. This is a device-dependent operation. For guidance in initializing the device, refer to the hardware reference manual for the device.

3. Test to see if device initialization was successful. If it was not, return the value zero.

4. Obtain the device granularity. For some devices, only one granularity is possible, while for others several granularities are possible. This is a device-dependent issue that is explained in the hardware reference manual for your device.

5. Return the device granularity.

DEVICE$READ PROCEDURE

The DEVICE$READ procedure must present the following PL/M-86 interface to the Bootstrap Loader:

```
DEVICE$READ: PROCEDURE (unit, blk$num, buf$ptr) PUBLIC;
          DECLARE   unit          WORD,
                    blk$num       DWORD,
                    buf$ptr       POINTER;
      .
      . (code)
      .
END DEVICE$READ;
```

where:

unit	The device's unit number, as defined during configuration.
blk$num	A 32-bit number specifying the number of the block that the Bootstrap Loader wants the procedure to read.
buf$ptr	A 32-bit POINTER to the buffer that is to receive the information from the secondary storage device.

The DEVICE$READ procedure does not return a value to the caller.

The following outline shows the steps that the DEVICE$READ procedure must perform to be compatible with the Bootstrap Loader:

1. Read the block specified by the blk$num parameters from the bootstrap device specified by the unit parameter into the memory location specified by the buf$ptr parameter.

2. Check for I/O errors. If none occurred, return to the caller. Otherwise, combine the device code, if any, for the device with 01 (in the form <device code>01), push the resulting word value onto the stack, and call the BSERROR procedure. For example, if the device code is 0B3H, push B301H onto the stack. If there isn't a device code, use 00.

   In PL/M-86, adding the following statements will accomplish this:

   ```
   DECLARE BSERROR EXTERNAL;
   DECLARE IO_ERROR LITERALLY '0B301H';

   CALL BSERROR(IO_ERROR);
   ```

   If you are calling the BSERROR procedure from assembly language, note that BSERROR follows the PL/M-86 LARGE model of segmentation; that is, declare BSERROR as:

   ```
   extrn     bserror:far
   ```

***

APPENDIX A
AUTOMATIC BOOT DEVICE RECOGNITION

Automatic boot device recognition allows the iRMX 86 Operating System to recognize the device from which it was bootstrap loaded and to assign a logical name (normally :SD:) to represent that device.

If you use this feature, you can configure versions of the Operating System that are device independent, that is, versions you can load and run from any device your system supports.

This section describes the automatic boot device recognition feature in detail. It consolidates the information found in the other iRMX 86 manuals and answers the following questions:

- How does automatic boot device recognition work?
- How do you configure a version of the Operating System that includes this feature?

HOW AUTOMATIC BOOT DEVICE RECOGNITION WORKS

The Nucleus, the Extended I/O System, and the second stage of the Bootstrap Loader combine to provide the automatic boot device recognition feature, as follows:

1. The second stage of the Bootstrap Loader, after loading the Operating System, places a pointer in the DI:SI register pair. This pointer points to a string containing the name of device from which the system was loaded. The name it uses is the one you (or whoever performed the configuration) supplied as a parameter in the %DEVICE macro when configuring the Bootstrap Loader.

2. The second stage sets the CX and DX registers to the value 1234H. This value signifies that the pointer contained in the DI:SI register pair is valid.

3. The root job checks CX and DX and then, if both contain 1234H, uses the pointer in DI:SI to obtain the device name. The root job sets a Boolean variable to indicate whether it found the name of the boot device.

4. The Nucleus checks the root job's Boolean variable and, if it is true (equal to 0FFH), places the device name in a segment and catalogs that segment in the root job's object directory under the name RQBOOTED.

5. The Extended I/O System looks up the name RQBOOTED and, if successful, obtains the device name from the segment cataloged there. If the name RQBOOTED is not cataloged in the root directory, the Extended I/O System uses a default device name that you must have specified during the configuration of the Extended I/O System (DPN prompt of the "EIOS" screen).

6. The Extended I/O System attaches the device as the system device, assigning it the logical name that you must have specified during the configuration of the Extended I/O System (DLN prompt on the "EIOS" screen).

HOW TO INCLUDE AUTOMATIC BOOT DEVICE RECOGNITION IN YOUR SYSTEM

This section describes the operations you must perform to include the automatic boot device recognition feature in your application. The operations include:

- The "EIOS" screen (see Figure A-1) contains several prompts that affect the automatic boot device recognition feature. They are: ABR, DLN, DPN, DFD, and DO.

  With the ABR prompt, you specify whether you want to include the automatic boot device recognition feature in your system. If you set ABR to "yes," the Extended I/O System automatically attaches the system device using the characteristics specified in the DLN, DFD, and DO prompts. You must not supply this information later in the "Logical Names" screen.

  If you set ABR to "no," the Extended I/O System does not attach a system device. In this case, the ICU does not display the DLN, DPN, DFD, and DO prompts.

```
      EIOS
      (ASC) All Sys Calls in EIOS                                Req
---> (ABR) Automatic Boot Device Recognition [Yes/No]           Yes
---> (DLN) Default System Device Logical Name [1-12 characters] SD
---> (DPN) Default System Device Physical Name [1-12 characters] w0
---> (DFD) Default System Device File Driver [Phys/Str/Named]   Named
---> (DO)  Default System Device Owners ID [0-0FFFFH]           0000H
      (EBS) Internal Buffer Size [0-0FFFFh]                      0400H
      (DDS) Default IO Job Directory Size [5-0FF0h]              0032H
      (ITP) Internal EIOS Task's Priorities [0-FFH]              0083H
      (PMI) EIOS Pool Minimum [0-0FFFFH]                         0180H
      (PMA) EIOS Pool Maximum [0-0FFFFH]                         0180H
      (EIR) Extended I/O System in ROM [Yes/No]                  No
```

Figure A-1. EIOS Configuration Screen (ABR, DLN, DPN, DFD, and DO)

With the DLN prompt, you can specify the logical name for your system device. If you change this value from the default (SD), you must change all other references to the :SD: logical name to the new name you specify. The Extended I/O System creates the logical name you specify only if you set ABR to "yes."

With the DPN prompt, you specify the physical name of a device that you want to use as your system device in case the Extended I/O System cannot find the name RQBOOTED cataloged in the root object directory. This situation normally occurs when you load your system using a means other than the Bootstrap Loader. For example, if you transfer the Operating System to your target system via the iSBC 957B load package or iSDM 86 or iSDM 286 monitor, there is no bootstrap device. In this case, the Extended I/O System uses the device name specified in the DPN prompt as the system device.

With the DFD and DO prompts, you set other characteristics associated with the system device. For most cases, the defaults (DFD=Named and DO=0000H) are the preferred values.

- During configuration of the Basic I/O System, you must specify device-unit information for the devices you wish to support. One of the prompts on each "Device-Unit Information" screen (NAM) requires you to specify the name of the device-unit. Another parameter (UN) requires you to specify the unit number. (See Figure A-2 for an example of these prompts.) To enable the automatic boot device recognition feature to work correctly, assign device-unit names and unit numbers that match the device names and unit numbers assigned during the configuration of the Bootstrap Loader.

- You assign the Bootstrap Loader device names and unit numbers by including or modifying %DEVICE macros in the first-stage configuration file (BS1.A86) or in the iSBC 957B configuration file. With the ICU, you can define device-unit names and unit numbers other than those that are valid for the Bootstrap Loader. But each Bootstrap Loader device name must have a corresponding device-unit name, and the unit numbers must be the same.

Before you can use the automatic boot device recognition feature, you must format your system device using the Release 6 version of the FORMAT command. The iRMX 86 CONFIGURATION GUIDE describes how to set up your system device for use with Release 6.

HOW TO EXCLUDE AUTOMATIC BOOT DEVICE RECOGNITION

To configure a system that does not include the automatic boot device recognition feature, set the ABR prompt in the "EIOS" screen to "No" (see Figure A-1). This disables the automatic boot device recognition feature.

```
    Intel iSBC 215/iSBX 218 Device-Unit Information
--->(NAM) Device-Unit Name [1-13 chars]
    (PFD) Physical File Driver Required [Yes/No]               Yes
    (NFD) Named File Driver Required [Yes/No]                  Yes
    (SDD) Single or Double Density Disks [Single/Double]       Single
    (SDS) Single or Double Sided Disks [Single/Double]         Single
    (EFI) 8 or 5 inch Disks [8/5]                              8
    (GRA) Granularity [0-0FFFFH]                               0080H
    (DSZ) Device Size [0-0FFFFFFFFH]                           0003E900H
--->(UN)  Unit Number on this Device [0-0FFH]                  0000H
    (UIN) Unit Info Name [1-17 Chars]
    (UDT) Update Timeout [0-0FFFFH]                            0064H
    (NB)  Number of Buffers [nonrand = 0/rand = 1-0FFFFH]      0006H
    (FUP) Fixed Update [True/False]                            True
    (MB)  Max Buffers [0-0FFH]                                 00FFH
```

Figure A-2. Device-Unit Information Screen (NAM and UN)

When you set ABR to "No", the ICU deletes the DLN, DPN, DFD, and DO prompts from the EIOS screen. Therefore, you must provide this information as input to the "Logical Names" screen. Figure A-3 shows an example of this screen after it has been filled in to include a logical name for the system device. The underlined information in Figure A-3 is the information you would supply if you set the ABR prompt in Figure A-1 to "No" and you want the system device to be a flexible diskette drive controlled by an iSBC 208 board.

```
    Logical Names
    Logical Name : logical_name,device_name,file_driver,owners-id
           [1-12 Chars, 1-14 Chars, Physical/Stream/Named, 0-0FFFFH]

    Logical Name : BB, BB, Physical, 0000H
    Logical Name : STREAM, STREAM, Stream, 0000H
--->Logical Name : SD, AF0, Named, 0000H
```

Figure A-3. Logical Names Screen

***

APPENDIX B PROMMING THE BOOTSTRAP LOADER WITH A SYSTEM DEBUG MONITOR

In Chapter 3, it was mentioned that one of the ways in which you can prepare to use the Bootstrap Loader is to combine it with one of the Intel monitor packages and burn the combined code into PROM. This appendix supplies information that is not contained in the reference manuals for the iSDM 86 and iSDM 286 monitors.

COMBINING WITH THE iSDM 86 SYSTEM DEBUG MONITOR

The iSDM 86 SYSTEM DEBUG MONITOR REFERENCE MANUAL correctly describes the procedure for combining the iRMX 86 Bootstrap Loader with the iSDM 86 monitor. However, if you are going to combine the Release 6 version of the Bootstrap Loader with the iSDM 86 monitor, you must first modify one of the files described in that manual. This file, called SDMGNB.CSD, must be changed to the read as is shown in Figure B-1. In the figure, the lines that are different from the listing in the iSDM 86 manual are marked with comments on the right-hand side of the figure.

COMBINING WITH THE iSDM 286 SYSTEM DEBUG MONITOR

The iSDM 286 SYSTEM DEBUG MONITOR REFERENCE MANUAL does not describe the procedure for combining the iRMX 86 Bootstrap Loader with the iSDM 286 monitor. This section gives the instructions required to burn the first stage and the iSDM 286 monitor into two 2764 EPROMs. You can modify this example to suit your own purposes, or you can follow it exactly. The step-by-step procedure is as follows:

Enter the name of the (version 1.3 or later) software used with the iUPP Universal Prom Programmer:

```
:f0:ipps
```

Specify that the PROMs are 2764 EPROMs:

```
type 2764
```

Initialize the file type to be loaded:

```
initialize 86
```

This says that the load file is an 8086 Object Module Format file.

```
run
;
asm86 :fl:bsl.a86 macro(90)                 ;changed line
asm86 :fl:bserr.a86 macro(50)
;asm86 :fl:b204.a86 macro(50)
;asm86 :fl:b206.a86 macro(50)
;asm86 :fl:b208.a86 macro(50)
;asm86 :fl:b215.a86 macro(50)
;asm86 :fl:b218a.a86 macro(50)              ;new line
;asm86 :fl:b251.a86 macro(50)               ;new line
;asm86 :fl:b254.a86 macro(50)
;asm86 :fl:bsasi.a86 macro(50)              ;new line
;asm86 :fl:bscsi.a86 macro(50)              ;new line
;
link86 &
     :fl:sdm86.lib(monitor), &
     :fl:%0.obj, &
     :fl:bsl.obj, &
     :fl:bserr.obj, &
     :fl:sdm86.obj, &
&    :fl:b204.obj, &
&    :fl:b206.obj, &
&    :fl:b208.obj, &
&    :fl:b215.obj, &
&    :fl:b218a.obj, &                       ;new line
&    :fl:b251.obj, &                        ;new line
&    :fl:b254.obj, &
&    :fl:bsasi.obj, &                       ;new line
&    :fl:bscsi.obj, &                       ;new line
     :fl:bsl.lib, &
     8087.lib to :fl:%0.lnk
;
loc86 &
     :fl:%0.lnk &
     addresses(classes(sdm86_data(400h), &
     stack(3c000h), &
     code(0f8000h))) &
     order(classes(sdm86_data, sdm86_stack, &
                   stack, data, boot, &
                   code)) &
     segsize(boot(1800h)) &
     start(%1) bootstrap noinitcode
;
exit
```

Figure B-1. Contents Of SDMGNB.CSD

Specify that the even-numbered bytes of the BS1 (first stage) file are to go into EPROM 0 and the odd-numbered bytes are to go into EPROM 1. (The address FD800H is an example value for a particular configuration. The numbers 3, 2, and 1 match ipps prompts for defining the information.)

```
format :fl:bsl(FD800H)
3
2
1
0 to :fl:bsl.evn
1 to :fl:bsl.odd
<cr>
```

Tell the software to program one EPROM with even-addressed bytes. (The address 0C00H is correct for the Bootstrap Loader-iSDM 286 combination.)

```
copy :fl:mbsl.evn to prom(0C00h)
```

Do the same thing for the odd-numbered bytes.

```
copy :fl:mbsl.odd to prom (0C00h)
```

Exit the ipps program.

```
exit
```

***

INDEX

Underscored entries are primary references.