intel

EXTENDED iRMX® II.3 OPERATING SYSTEM DOCUMENTATION

VOLUME 4 SYSTEM UTILITIES AND PROGRAMMING INFORMATION

Order Number: 461847-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are located directly before the reader reply card in the back of the manual.

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 or 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.

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

Above	iLBX	iPSC	OpenNET
BITBUS	i_m	iRMX	ONCE
COMMputer	iMDDX	iSBC	Plug-A-Bubble
CREDIT	iMMX	iSBX	PROMPT
Data Pipeline	Insite	iSDM	Promware
Genius	int_el	iSSB	QUEST
i	int_elBOS	iSXM	QueX
ı	Intelevision	Library Manager	Ripplemode
i^2ICE	int_eligent Identifier	MCS	RMX/80
ICE	int_eligent Programming	Megachassis	RUPI
iCEL	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SLD
iDBP	iOSP	MULTICHANNEL	UPI
iDIS	iPDS	MULTIMODULE	VLSiCEL
	iPSB		

XENIX, MS-DOS, Multiplan, and Microsoft are trademarks of Microsoft Corporation. UNIX is a trademark of Bell Laboratories. Ethernet is a trademark of Xerox Corporation. Centronics is a trademark of Centronics Data Computer Corporation. Chassis Trak is a trademark of General Devices Company, Inc. VAX and VMS are trademarks of Digital Equipment Corporation. Smartmodem 1200 and Hayes are trademarks of Hayes Microcomputer Products, Inc. IBM is a registered trademark of International Business Machines. Soft-Scope is a registered trademark of Concurrent Sciences.

Intel®

VOLUME PREFACE

MANUALS IN THIS VOLUME

This volume (Volume 4, *Extended iRMX® II Operating System Utilities and Programming Information*) includes the following manuals:

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

The *Extended iRMX® II Bootstrap Loader Reference Manual* describes the use of the Bootstrap Loader and how to modify the Bootstrap Loader files.

The *Extended iRMX® II System Debugger Reference Manual* describes the iRMX® II Operating System's static debugger. All of the System Debugger commands are explained and an example debugging session is provided.

The *Extended iRMX® II Disk Verification Utility Reference Manual* describes the Disk Verification Utility which is used to examine and modify the data structures of iRMX® II named and physical volumes.

The *Guide to the Extended iRMX® II Interactive Configuration Utility* manual describes the Interactive Configuration Utility commands and menus, and provides an example system configuration.

The *Extended iRMX® II Programming Techniques Reference Manual* provides programming techniques and examples.

VOLUME CONTENTS

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

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

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

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

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

VOLUME 3: *Extended iRMX® II System Calls*

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

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

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

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

Extended iRMX® II Interactive Configuration Utility Reference Manual

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

intel

EXTENDED iRMX® II BOOTSTRAP LOADER REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

INTRODUCTION

The Bootstrap Loader enables you to generate a system that can bootload from Intel-supplied or custom devices. A bootable system gains control immediately after power-up or system reset. This manual provides information that enables you to configure your system to boot from specific devices, to include your own custom device drivers as part of the system, and to place your generated system into PROM devices.

READER LEVEL

The manual assumes that you are familiar with the iRMX II Operating System and an editor with which you can edit source code files. It may also be helpful if you are familiar with the following:

- SUBMIT files.
- ASM86 source code files.

MANUAL OVERVIEW

This manual is organized as follows:

Chapter 1	This chapter provides an overview of the Bootstrap Loader operations.
Chapter 2	This chapter provides an operator's viewpoint of using the Bootstrap Loader.
Chapter 3	This chapter describes how to configure the first stage of the Bootstrap Loader.
Chapter 4	This chapter describes how to configure the third stage of the Bootstrap Loader.
Chapter 5	This chapter describes how to write custom first-stage drivers.
Chapter 6	This chapter describes how to write custom third-stage drivers.

Chapter 7	This chapter describes error handling procedures.
Appendix A	This appendix describes how to include automatic boot device recognition into your system.
Appendix B	This appendix describes how to load the Bootstrap Loader and the monitor into PROM devices.

CONVENTIONS

The following conventions are used throughout this manual:

- Information appearing as UPPERCASE characters when shown in keyboard examples must be entered or coded exactly as shown. You may, however, mix lower and uppercase characters when entering the text.
- Fields appearing as lowercase characters within angle brackets (< >) when shown in keyboard examples indicate variable information. You must enter an appropriate value or symbol for variable fields.
- User input appears in one of the following forms:

```
as bolded text within a screen
```

- The term "iRMX II" refers to the Extended iRMX II.3 Operating System.
- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- All numbers, unless otherwise stated, are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

Intel®

CONTENTS

CHAPTER 1 **PAGE**
OVERVIEW OF BOOTSTRAP LOADER OPERATIONS

1.1 Introduction to the Bootstrap Loader 1-1
1.2 The Stages of the Bootstrap Loader 1-2
1.2.1 First Stage 1-2
1.2.2 Second Stage 1-3
1.2.3 Third Stage 1-4
1.2.3.1 Generic Third Stage 1-4
1.2.3.2 Device-Specific Third Stage 1-5
1.2.3.3 Naming the Third Stage 1-5
1.2.4 Load File 1-5
1.3 Device Drivers 1-7
1.4 Memory Locations Used by the Bootstrap Loader 1-8
1.5 Configuring Your Own Bootstrap Loader 1-9

CHAPTER 2 **PAGE**
USING THE BOOTSTRAP LOADER

2.1 Introduction 2-1
2.2 Operator's Role in Bootstrap Loading 2-1
2.2.1 Specifying the Load File 2-1
2.2.2 Debug Option 2-3
2.3 Placing the Bootstrap Loader Into Memory 2-4
2.4 Choosing a Third Stage 2-6

CHAPTER 3 **PAGE**
CONFIGURING THE FIRST STAGE

3.1 Introduction 3-1
3.2 BS1.A86 and BS1MB2.A86 Configuration Files 3-2
3.2.1 %BIST Macro (MULTIBUS® II Only) 3-8
3.2.2 %COPY Macro (MULTIBUS® II Only) 3-9
3.2.3 %AUTO_CONFIGURE_MEMORY Macro (MULTIBUS® II Only) 3-10
3.2.4 %CPU Macro 3-11
3.2.5 %BMPS Macro (MULTIBUS® II Only) 3-11
3.2.6 %iAPX_186_INIT Macro (iRMX® I MULTIBUS® I Systems Only) 3-13
3.2.7 %CONSOLE, %MANUAL, and %AUTO Macros 3-14
3.2.8 %LOADFILE Macro 3-16
3.2.9 %DEFAULTFILE Macro 3-17

CHAPTER 3 (continued) **PAGE**

3.2.10 %RETRIES Macro 3-17
3.2.11 %CLEAR_SDM_EXTENSIONS Macro 3-18
3.2.12 %CICO Macro 3-18
3.2.13 %SERIAL_CHANNEL Macro 3-19
3.2.14 %DEVICE Macro 3-24
3.2.15 %END Macro 3-27
3.3 BSERR.A86 Configuration File 3-27
3.3.1 %CONSOLE Macro 3-28
3.3.2 %TEXT Macro 3-28
3.3.3 %LIST Macro 3-28
3.3.4 %AGAIN Macro 3-29
3.3.5 %INT1 Macro 3-29
3.3.6 %INT3 Macro 3-30
3.3.7 %HALT Macro 3-30
3.3.8 %END Macro 3-31
3.4 Device Driver Configuration Files 3-31
3.4.1 %B208 Macro 3-32
3.4.2 %BMSC and %B220 Macros 3-32
3.4.3 %B218A Macro 3-33
3.4.4 %B224A Macro 3-34
3.4.5 %B251 Macro 3-35
3.4.6 %B254 Macro 3-36
3.4.7 %B264 Macro 3-36
3.4.8 %BSCSI Macro 3-37
3.4.9 %SASI_UNIT_INFO Macro 3-39
3.4.10 User-Supplied Drivers 3-40
3.5 Generating the First Stage 3-41
3.5.1 Modifying the BS1.CSD Submit File 3-44
3.5.2 Invoking the BS1.CSD Submit File 3-46
3.6 Memory Locations of the First and Second Stages 3-47

CHAPTER 4 **PAGE**
CONFIGURING THE THIRD STAGE

4.1 Introduction 4-1
4.2 Overview of Third Stage Configuration 4-1
4.3 BS3.A86, BS3MB2.A86 and BG3.A86 Configuration Files 4-2
4.3.1 %BMPS Macro (MULTIBUS® II Only) 4-5
4.3.2 %DEVICE Macro (BS3.A86 and BS3MB2.A86 Only) 4-6
4.3.3 %SASI_UNIT_INFO Macro (BSCSI.A86 File) 4-7
4.3.4 %INT1 Macro 4-9
4.3.5 %INT3 Macro 4-10
4.3.6 %HALT Macro 4-10
4.3.7 %CPU_BOARD Macro 4-10

CHAPTER 4 (continued) **PAGE**

4.3.8 %INSTALLATION Macro (BG3.A86 Only) ... 4-11
4.3.9 %END Macro ... 4-12
4.3.10 User-Supplied Drivers ... 4-12
4.4 Generating the Third Stage ... 4-13
4.4.1 Modifying the Submit Files ... 4-14
4.4.2 Invoking the Submit File ... 4-15
4.5 Memory Locations of the Three Stages ... 4-16

CHAPTER 5 **PAGE**
WRITING A CUSTOM FIRST-STAGE DRIVER

5.1 Introduction ... 5-1
5.2 Device Initialize Procedure ... 5-2
5.3 Device Read Procedure ... 5-3
5.4 Supplying Configuration Information to the First-Stage Driver ... 5-4
5.4.1 Hard-Coding the Configuration Information ... 5-4
5.4.2 Providing a Configuration File ... 5-5
5.5 Using the MULTIBUS® II Transport Protocol ... 5-8
5.5.1 Message Passing Controller Initialization ... 5-9
5.5.2 Message Types ... 5-10
5.5.3 Request/Response Transaction Model ... 5-10
5.5.4 Send and Receive Transaction Models ... 5-15
5.5.5 Message Broadcasting ... 5-20
5.5.6 Transmission Modes ... 5-22
5.5.7 Interconnect Space ... 5-22
5.5.8 Driver Code Considerations ... 5-30
5.6 Changing BS1.A86 or BS1MB2.A86 to Include the New First-Stage Driver ... 5-33
5.7 Generating a New First Stage Containing the Custom Device Driver ... 5-34

CHAPTER 6 **PAGE**
WRITING A CUSTOM THIRD-STAGE DRIVER

6.1 Introduction ... 6-1
6.2 What a Third-Stage Device Driver Must Contain ... 6-1
6.3 Device Initialization Procedure ... 6-3
6.4 Device Read Procedure ... 6-4
6.5 Protected Mode Considerations ... 6-6
6.6 Supplying Configuration Information to the Third-Stage Driver ... 6-7
6.7 Using MULTIBUS® II Transport Protocol ... 6-8
6.8 Changing BS3.A86 to Include the New Third-Stage Driver ... 6-8
6.9 Generating a New Third Stage Containing the Custom Driver ... 6-9

CHAPTER 7 **PAGE**
ERROR HANDLING
7.1 Introduction 7-1
7.2 Analyzing Bootstrap Loader Failures 7-1
7.2.1 Actions Taken by the Bootstrap Loader After an Error 7-1
7.2.2 Analyzing Errors With Displayed Error Messages 7-2
7.2.3 Analyzing Errors Without Displayed Error Messages 7-5
7.2.4 Initialization Errors 7-7

APPENDIX A **PAGE**
AUTOMATIC BOOT DEVICE RECOGNITION
A.1 Introduction A-1
A.2 How Automatic Boot Device Recognition Works A-1
A.3 How to Include Automatic Boot Device Recognition A-2
A.4 How to Exclude Automatic Boot Device Recognition A-5

APPENDIX B **PAGE**
PROMMING THE BOOTSTRAP LOADER AND THE ISDM™ MONITOR
B.1 Introduction B-1
B.2 Incorporating the iSDM™ Monitor B-1

TABLES

TABLE		PAGE
1-1	Intel-Supplied Bootstrap Loader Drivers	1-8
2-1	Supplied Third Stage Files	2-7
3-1	Procedure Names for Intel-Supplied First Stage Drivers	3-25
3-2	5.25-Inch Diskettes Supported by iSBC® 208 and MSC-Specific Drivers	3-26
3-3	8-Inch Diskettes Supported by iSBC® 208 and MSC-Specific Drivers	3-26
4-1	Names for Intel-Supplied Third Stage Drivers	4-7
4-2	Memory Locations Used by the Bootstrap Loader	4-16
7-1	Postmortem Analysis of Bootstrap Loader Failure	7-6

FIGURES

FIGURE		PAGE
3-1	Intel-Supplied BS1.A86 File	3-3
3-2	Intel-Supplied BS1MB2.A86 File	3-6
3-3	First Stage Configuration File BSERR.A86	3-27
3-4	First Stage Configuration File BS1.CSD	3-42
3-5	Excluding the iSBC® 251 and iSBC® 254 Drivers	3-45
4-1	Intel-Supplied BS3.A86 File	4-3
4-2	Intel-Supplied BS3MB2.A86 File	4-4
4-3	Intel-Supplied BG3.A86 File	4-5
4-4	Device-Specific Third Stage SUBMIT File (BS3.CSD)	4-13
4-5	Generic Third Stage SUBMIT File (BG3.CSD)	4-14
5-1	Hard-Coded Configuration Information	5-5
5-2	Modified BS1.CSD File	5-8
5-3	Modified BS1.A86 File	5-34
6-1	Changing the BS3.A86 File	6-9
A-1	EIOS Configuration Screen (ABR)	A-2
A-2	ABDR Screen (DLN, DPN, DFD, DO)	A-3
A-3	Device-Unit Information Screen (NAM and UN)	A-4
A-4	Logical Names Screen	A-5

CHAPTER 1
OVERVIEW OF BOOTSTRAP LOADER OPERATIONS

1.1 INTRODUCTION

The Bootstrap Loader is a program that is not part of any particular Operating System. Rather, it is a program that loads an application system into RAM from secondary storage so that it can begin running. This process is called bootstrap loading or booting. Booting can occur when the system is turned on, when the system is reset, or under operator control when the monitor is active.

The Bootstrap Loader eliminates the need to place complete applications into PROM devices. Instead, you can place the Bootstrap Loader--a relatively small program--into PROM devices and store your application system on a mass storage device. The Bootstrap Loader can then be used to load the application program into RAM.

The Bootstrap Loader consists of three stages.

The first stage resides in PROM devices. It determines the name of the file to load, loads part of the second stage, and passes control to that part. Intel System 300 Series Microcomputers are delivered with the first stage of the Bootstrap Loader and the iSDM monitor already placed in PROM devices. Intel Modules Development Platforms are delivered similarly except with the D-MON386 monitor. If you are building your own computer systems, you can use the information in this manual to configure a first stage and place it into PROM devices.

The second stage resides on track 0 of every iRMX-formatted named volume. That is, whenever you use the Human Interface FORMAT command to format a volume, the second stage is copied to that volume. When invoked, the second stage finishes loading itself into memory and then loads a file from the same volume and passes control to it. The contents of this load file depend on the kind of system you are loading. If you are loading an iRMX I system, the file loaded by the second stage contains the application system itself. If you are loading an iRMX II system, the file loaded by the second stage contains the third stage of the Bootstrap Loader, which finishes the loading process.

The third stage is necessary for loading iRMX II applications, because these applications require the 80286 processor to be running in protected mode and because they use the OMF-286 object module format. The OMF-286 format is different from the OMF-86 format and therefore cannot be handled by the second stage. The third stage places the processor in protected mode, loads the iRMX II application system, and transfers control to that application system. The third stage resides in a named file on the same volume as the second stage. Your Bootstrap Loader package contains a configured third stage that can load applications from selected devices. The instructions in this manual can help you configure your own third stage to add support for other devices.

The bootstrap loading process cannot be completed without a device driver. The device driver is a small program that provides the interface between the Bootstrap Loader and a hardware device (or a controller for the device). When you configure the Bootstrap Loader (a task that is independent of operating system configuration), you specify the device drivers that the Bootstrap Loader requires. During the course of configuration, these device drivers (which are usually distinct from the drivers needed by the application system) are linked to the Bootstrap Loader automatically.

1.2 THE STAGES OF THE BOOTSTRAP LOADER

The Bootstrap Loader has a number of stages that control the loading of the application system. iRMX I applications load with a two-stage process. iRMX II applications use these two stages but also require a third stage.

1.2.1 First Stage

The Bootstrap Loader's first stage consists of two parts. One part is the code for the first stage, and the other part is a set of minimal device drivers used by the first and second stages to initialize and read from the device that contains the system to be booted.

The Bootstrap Loader package contains device drivers for many common Intel devices. To support other devices, you can write your own drivers and configure them into the first stage.

To use the Bootstrap Loader, the first stage must be in one of two places. The natural place for the first stage is in PROM devices, either as a standalone product or combined with a monitor. Intel System 310 and 380 Series Microcomputers are delivered with the Bootstrap Loader's first stage, the iSDM monitor, and the System Confidence Test (SCT) in the PROM devices. Intel System 320 Series Microcomputers are delivered with the Bootstrap Loader's first stage, the iSDM monitor, the D-MON386 monitor, and the SCT in the PROM devices. Intel Modules Development Platforms are delivered with the Bootstrap Loader's first stage, the D-MON386 monitor, and the SCT in the PROM devices.

If you have a system that includes the iSDM monitor and you are adding your own device driver to the Bootstrap Loader's first stage, you might find it useful to load the first stage into the target system's RAM using a development system iSDM loader and activate the first stage under iSDM control from the development system. After activating the first stage, you could then debug driver code. If your system includes the D-MON386 monitor, however, you must perform all driver debugging from the target system. You cannot download the first stage from a development system into the target system and then use D-MON386 to initiate program execution. When debugging under these latter circumstances, you may wish to either debug within the PROM devices or perhaps use a working Bootstrap Loader to bootload the Bootstrap Loader that contains the new driver.

When the first stage begins running, it first identifies the bootstrap device and the name of the file to boot, either by accepting that information from a command line entered at the monitor or by using default characteristics established when the first stage was configured. The Bootstrap Loader next calls its internal device driver for the device, which initializes the device and reads the first portion of the second stage into memory. (The second stage always resides on track 0 and block 0 of the named volume, so it can be accessed easily by the first stage.) After calling the internal device driver, the first stage passes control to the second stage.

Because the first stage works on both 8086/186- and 80286/386-based computers, it operates in real address mode when running in an 80286/386-based system. This means that any device drivers you write for the first stage must also operate in real address mode.

1.2.2 Second Stage

Unlike the first stage, the second stage of the Bootstrap Loader is not configurable. Its size is always the same (less than 8K bytes), and it does not depend on the application in any way. The code for the second stage resides on all volumes formatted with the iRMX I or iRMX II Human Interface FORMAT commands. Therefore, the second stage is always available for loading applications residing on random access devices.

When the second stage receives control, it finishes loading itself into memory and then loads the file determined by the first stage. When loading the file, it uses the same device driver used by the first stage. In iRMX I systems, the load file is the application system itself. In iRMX II systems, this file is the third stage of the Bootstrap Loader.

NOTE

You cannot bootstrap load the iRMX II.1, II.2, II.3 Operating System from a volume that was formatted using the iRMX I.6 or I.7 (iRMX 86 Release 6 or 7) FORMAT command. However, you can make the volume bootable without reformatting the entire volume and losing the data stored on it. To be able to boot both the iRMX I and iRMX II Operating Systems from the same volume, invoke the iRMX II.3 FORMAT command and specify the BOOTSTRAP control. With BOOTSTRAP specified, FORMAT just replaces the second stage on track 0 of the volume while leaving the remaining data untouched. When the FORMAT command finishes, you can bootstrap load both the iRMX I and iRMX II Operating Systems from the same volume.

1.2.3 Third Stage

The third stage of the Bootstrap Loader is used for loading iRMX II-based applications into memory. The third stage resides in a named file on the bootstrap device. Both the third stage and the application system to be loaded must reside in the same directory on the volume.

There are two types of third stages: a generic third stage and a device-specific third stage. The type needed for your system depends on the size of the application system you intend to load.

1.2.3.1 Generic Third Stage

The generic third stage is so named because it can load application systems from any device that the first stage recognizes. This stage contains no device driver of its own. Instead, it uses the same device driver used by the first and second stages. This means that you won't need to write a separate device driver to work in protected mode, but it also means that the generic third stage runs in real address mode. In real address mode, addressability is restricted to the first (lowest) megabyte of memory. Therefore, the generic third stage can load only those application systems that are smaller than 840K bytes. The remaining space is used by the Bootstrap Loader, the monitor and the SCT. To load larger applications, you must use a device-specific third stage.

When the generic third stage receives control, it uses the device driver supplied in the first stage to load the application system. Then it switches the processor into protected virtual address mode and passes control to the application.

1.2.3.2 Device-Specific Third Stage

The device-specific third stage switches the processor to protected virtual address mode before loading the application system. This enables this stage to load into memory addresses higher than one megabyte. However, because this stage switches the processor into protected mode, it cannot use the first stage's device drivers (which operate only in real mode). Instead, it must contain its own device driver, operating in protected mode, to control the device from which the application system is loaded.

The device-specific third stage supplied in your Bootstrap Loader package supports the following devices:

- iSBC 215G/iSBX 218A winchester and diskette controller combination or the iSBC 214 controller
- iSBC 264 bubble memory controller
- iSBC 186/224A multi-peripheral controller
- SCSI (Small Computer Systems Interface) and SASI (Shugart Associates Systems Interface) peripheral bus controllers having iSBC 286/100A CPU board as the host.

If you want to boot from any other device, you must write a protected mode device driver for the device and link the driver in when you configure the device specific third stage (see Chapter 6).

When the device-specific third stage receives control, it performs the same operations as the generic third stage. However, before invoking the device driver to load the application system, it switches the processor into protected mode. This enables the third stage to load applications that reside outside the first megabyte.

1.2.3.3 Naming the Third Stage

Both the generic and the device-specific third stages are stored as executable files. The base portion of this file's name -- the filename minus any extension -- must be the same as the base portion of the file containing the application system to be loaded. Because the name of the third stage and the name of the application system must match, you must provide a separate third stage file for each bootable system on the volume. To provide additional third stage files, simply make a copy of the third stage file you are currently using. Name the copy so that it matches the application system you intend to load.

1.2.4 Load File

The load file is a file containing the application system you are trying to boot. The load file should be on an iRMX I- or iRMX II-formatted named volume. This volume must have been formatted by the Human Interface FORMAT command. If the load file is an iRMX II application, the volume must also have a file containing the third stage of the Bootstrap Loader.

If your load file is an iRMX II application, the name of that file must correspond to the name of the Bootstrap Loader third stage, as follows:

- The base portion of the load file's name (the filename minus the extension) must be the same as that of the file containing the third stage.
- The extension portion of the load file's name must consist of the characters ".286".

The following are examples of valid and invalid third stage/load file combinations:

Valid Combinations

Third stage --	MYSYS
Load file --	MYSYS.286
Third stage --	SYS1.3RD
Load file --	SYS1.286

Invalid Combinations

Third stage --	MYSYS
Load file --	YOURSYS.286
Third stage --	MYSYS.3RD
Load file --	MYSYS.LOD

When you configure the first stage of the Bootstrap Loader, you can choose the file name that will be used if the operator doesn't specify a filename when invoking the Bootstrap Loader. By default, the file name is /SYSTEM/RMX86 for iRMX I load files. For iRMX II systems, /SYSTEM/RMX86 is the default name of the Bootstrap Loader's third stage and /SYSTEM/RMX86.286 is the default name of the iRMX II load file.

NOTE

Because of the way the Bootstrap Loader interprets filenames, the only period (.) allowed in the entire pathname for the load file is the one that precedes the extension 286. For example, the pathname /SYSTEM.1/MYSYS.286 is invalid because it contains more than one period.

1.3 DEVICE DRIVERS

When the Bootstrap Loader starts running, there is no software in place to enable the processor to communicate with the device from which you want to load the system. Part of the task of the Bootstrap Loader is to establish communications with the boot device. To communicate with devices, the Bootstrap Loader must include programs, called device drivers, for the devices from which you want to boot. When configuring the Bootstrap Loader, you specify the device drivers you want to include. The configuration process links the drivers to the Bootstrap Loader code.

Both the first stage and the device-specific third stage require their own drivers. The first-stage drivers operate in real address mode and are used to load iRMX I applications and the third stage of the Bootstrap Loader. The generic third stage also uses the first-stage drivers to load iRMX II applications.

The third-stage drivers operate in protected virtual address mode and are used by the device-specific third stage to load iRMX II applications into the full 16 megabyte address space.

The first stage must include a real mode device driver for each device from which you want to boot. The generic third stage includes no drivers of its own, but the device-specific third stage must include a protected mode driver for each of the boot devices. Intel includes several real and protected mode drivers in the Bootstrap Loader package, as listed in Table 1-1. All the real mode drivers can be used with the first stage and with the generic third stage. All the protected mode drivers can be used with the device-specific third stage.

If you want to boot from a device not supported by these device drivers, you can write your own device driver. See Chapter 5 for information on writing a new device driver.

Table 1-1. Intel-Supplied Bootstrap Loader Drivers

Driver	Type
iSBC 208 Flexible Disk Drive Controller.	Real Mode. Also used with the generic third stage.
Mass Storage Controller (MSC), supporting the iSBC 214 and iSBC 215G controller boards. Also supports the iSBX 218A controller when it is mounted on the iSBC 215G board.	Both Real and Protected Mode.
iSBX 218A Flexible Disk Controller (used on a processor board)	Real Mode only. Also used with the generic third stage.
iSBC 220 SMD Disk Controller	Real Mode only. Also used with the generic third stage.
iSBC 186/224A	Both Real and Protected Mode.
iSBX 251 Bubble Memory Controller	Real Mode Only.
iSBC 254 Bubble Memory Controller	Real Mode Only.
iSBC 264 Bubble Memory Controller	Both Real and Protected Mode.
SCSI (Small Computer Systems Interface) and SASI (Shugart Associates Systems Interface) Peripheral Bus Controllers when the host for these controllers is the iSBC 286/100A CPU board.	Both Real and Protected Mode.
SCSI (Small Computer Systems Interface) and SASI (Shugart Associates Systems Interface) Peripheral Bus Controllers when the host for these controllers is the iSBC 186/03A CPU board.	Real Mode Only.

1.4 MEMORY LOCATIONS USED BY THE BOOTSTRAP LOADER

All three stages of the Bootstrap Loader reside in or are loaded into memory. This section discusses the memory locations for different types of systems.

NOTE

When you configure your own version of the Bootstrap Loader, you must ensure that the memory locations occupied by the three stages do not overlap. In addition, when you configure the application system, you must ensure that it will not be loaded into the memory occupied by the stage that is loading it. However, you can configure this memory so that the iRMX I and iRMX II free space manager has access to it once the application begins running.

The code for the first stage is normally located in PROM devices in the upper part of the memory address space. The first stage data and stack are located by in conjunction with the second stage code at address 0B8000H. The second stage uses the same data and stack as the first stage. The first stage data and stack plus the second stage code require 8K bytes of memory. You can change the locations of the first stage data and stack, and the second stage code by selecting a different address for the second stage when you invoke the SUBMIT file, BS1.CSD, to configure the first stage. Chapter 3 describes the BS1.CSD file.

The device-specific third stage is located by default at address 0BC000H. It requires 16K bytes of memory, and it uses its own stack and data segments. You can change the location of the device-specific third stage by using the BS3.CSD SUBMIT file to generate your own version. Chapter 4 describes the BS3.CSD file.

The generic third stage is located by default at address 0BC000H. Unlike the device-specific third stage, it uses the data and stack of the first stage (because it uses the first-stage device drivers). You can change the location of the generic third stage by using the BG3.CSD SUBMIT file to generate your own version of it. Chapter 4 describes the BG3.CSD file.

When you use the second stage and generic third stage loaded into memory at their default addresses (0B8000H and 0BC000H), blocks of memory beginning at these two addresses are used to load the application. The generic third stage uses 16K bytes of memory. Thus, if your application were to occupy memory between 0B8000H and 0BFFFFH, the generic third stage would fail to load the application.

1.5 CONFIGURING YOUR OWN BOOTSTRAP LOADER

If you intend to create your own version of the Bootstrap Loader, you must use the Bootstrap Loader configuration and generation files supplied by Intel. In iRMX I systems, these files reside by default in the directory /RMX86/BOOT. In iRMX II systems, the files reside in the directory /RMX286/BOOT. Information about configuring the first and third stages is available in Chapters 3 and 4, and information on writing new device drivers is available in Chapters 5 and 6.

CHAPTER 2
USING THE BOOTSTRAP LOADER

2.1 INTRODUCTION

The procedure for using the Bootstrap Loader depends on where you locate the first stage, and for iRMX II users, which third stage you choose. This chapter explains the operator's role, methods of defining the first stage, and options to consider when choosing a third stage.

2.2 OPERATOR'S ROLE IN BOOTSTRAP LOADING

The operator's principal role in the bootstrap loading process is to specify the pathname of the file that is to be loaded. For iRMX I systems, this is the pathname of the application system. For iRMX II systems, this is the pathname of the Bootstrap Loader's third stage. If the operator is using the Intel-supplied first stage, the load file specifications can be entered in one of the following ways:

- By specifying neither the device name nor the file name
- By specifying both the device name and the file name
- By specifying the device name only
- By specifying the file name only

In addition, if you have the iSDM monitor, the operator can also use the Debug option to specify that control should pass to the monitor after loading is complete. (The D-MON386 monitor does not support a debug option.)

2.2.1 Specifying the Load File

An operator can specify a load file:

- When the monitor has issued a prompt. In this case, the operator can enter the monitor's B (bootstrap) command, followed by the name of the load file (include the name within single quotes if you are using the D-MON386 monitor). For this to work, the Bootstrap option must have been configured into the monitor. Refer to the *iSDM System Debug Monitor User's Guide* and the *D-MON386 Debug Monitor for the 80386 User's Guide* for information on configuring monitors.

- When the first stage of the Bootstrap Loader has issued an asterisk (*) prompt. When this prompt appears, the first stage waits for an operator to enter the name of the load file.

The method used to determine which file to load depends on the configuration of the Bootstrap Loader's first stage. Refer to Chapter 3 for more information about first stage configuration.

When entering the monitor's B command or responding to the Bootstrap Loader's asterisk prompt, the operator must specify the load file. One way to do this is to simply press Carriage Return. This causes the Bootstrap Loader to search for a default file on the default device (these defaults are set up when you configure the first stage). The Intel-supplied first stage uses the following pathname as its default:

/SYSTEM/RMX86

If you were using the default first stage and you wanted to load the file called /SYSTEM/RMX86 from the default device, you could simply type the B command with no parameters (if you boot from the monitor) and press Carriage Return, or type a Carriage Return only (if the Bootstrap Loader displays its own prompt).

If you need to specify a load file that is different from the default one, use the following format for the specification:

:device:pathname	(iSDM)
':device:pathname'	(D-MON386)

Where:

:device:	This is the name of the secondary storage device that contains the load file. If you omit the device name, the default device is used (as established during first stage configuration).
pathname	When loading iRMX I applications, this is the full pathname of the file you want to load. When loading iRMX II applications, this is the full pathname of the Bootstrap Loader's third stage. For iRMX II systems, the file to be loaded is assumed to have the same pathname as the third stage except for the filename extension, which is assumed to be .286. If you omit this name, the Bootstrap Loader attempts to load the default file (always /SYSTEM/RMX86).

To invoke the Bootstrap Loader with the monitor's B command, the processor must be running in real address mode. If your processor is running in real address mode, you can simply break to the monitor and issue the boot command.

However, if the processor is running in protected virtual address mode (as it is when the iRMX II Operating System is in control), you cannot boot another system by breaking to the monitor and issuing a boot command. You must first reset the system. After resetting the system, you can invoke the Bootstrap Loader at the monitor prompt.

Example 1: Assume that an iRMX I application system resides in the file /SYSTEM/MY86SYS on drive :WF0:. You can boot this system by issuing the following command at the iSDM monitor prompt:

```
.
```

Example 2: Assume that an iRMX II system resides in the file /SYSTEM/MYSYS.286 on drive :WF0:, and that the third stage of the Bootstrap Loader resides in the file /SYSTEM/MYSYS. If the processor is in real address mode, you can boot this system by issuing the following command at the D-MON386 monitor prompt:

```
>
```

2.2.2 Debug Option

Assuming that the iSDM monitor is present in the system, the operator can include a debug option when specifying a load file (the D-MON386 monitor does not support a debug option). This option instructs the Bootstrap Loader to do the following immediately after loading is complete:

- Set a breakpoint at the first instruction to be executed by the application system. For iRMX I systems, the breakpoint will be set in the load file. For iRMX II systems, the load file (the third stage) will be loaded as always and it will load the application system. The breakpoint will then be set in the application system.
- Pass control to the iSDM monitor, which displays an "Interrupt 3 at <xxxx:xxxx>" message at the terminal, issues its prompt (a single period for real-mode iRMX I systems, two periods for protected-mode iRMX II systems), and waits for a command from the terminal. At this point the operator can invoke any of the iSDM monitor commands that are appropriate for real or protected mode. (To continue running the loaded program, enter G<cr>.)

One advantage of the Debug switch is that the monitor's interrupt message tells you that the loading process was successful. If a system you are booting fails, you might not otherwise be able to tell whether the bootstrap load itself was unsuccessful, or whether the system loaded successfully and then failed during initialization. The presence or absence of the interrupt message when you use the Debug option clarifies whether the loading was successful.

Because the Debug option leaves you in the monitor, you can alter the contents of specific memory locations and perform other monitor actions (such as debugging) before you start your system running with the monitor's G command.

To use the Debug option when you are invoking the Bootstrap Loader from the iSDM monitor, include the letter D in the command line immediately after the B (boot) command. Specify any load file pathname after the B and D characters.

For example, any of the following command lines invoke the Bootstrap Loader (from the iSDM monitor) with the Debug option:

```
.
.
.
.
```

Notice that the "D" and any pathname must be separated by at least one space.

You can also use the Debug option on systems in which the Bootstrap Loader is configured to request the load file name; that is, on systems that issue the Bootstrap Loader's first stage asterisk (*) prompt. On these systems, place the "D" in the command line before the load file specification (separated by at least one space). Examples of this are:

```
*
*
*
*
```

2.3 PLACING THE BOOTSTRAP LOADER INTO MEMORY

Before you can invoke the Bootstrap Loader, you must place it into memory. Several ways exist to place the Bootstrap Loader into memory:

1. Place the first stage, configured for standalone operation, in PROM devices. In this case, the first stage begins to run on power-up or reset. Depending on its configuration, the standalone Bootstrap Loader may issue an asterisk prompt so that you can enter the name of the load file. To configure the first stage for standalone operation, refer to Chapter 3.

2. Configure the monitor to include the Bootstrap option, reconfigure the first stage of the Bootstrap Loader to include the first stage device driver(s) needed for bootstrap loading as not all of the device drivers supplied with the Bootstrap Loader will fit into the memory range provided by the monitor. Then program new PROM devices with the combination of the monitor and the first stage of the Bootstrap Loader. With this method, you initiate bootstrap loading via the monitor's 'B' (boot) command. To use this method, refer to chapter 3 for configuration information. Refer to the *Guide to the Extended iRMX II Interactive Configuration Utility* for information on programming a monitor and the Bootstrap Loader into the same set of PROM devices.

3. Place the first stage in secondary storage. Then, using the iSDM monitor or ICE in-circuit emulator, invoke the first stage. This procedure is particularly useful when you are adding a new device driver to the first stage and you need to debug the code. To configure the first stage for standalone operation where loading is to be performed with the iSDM monitor or ICE in-circuit emulator, refer to Chapter 3.

NOTE

If your system includes the D-MON386 monitor, you cannot download the first stage from one system to another and then invoke it using D-MON386 as described above. The previous description applies only to a system configured with the iSDM monitor or ICE in-circuit emulator.

4. Place the first stage in secondary storage, and then load it programmatically. This applies only to iRMX I systems. Because the iRMX II Operating System cannot switch back to real mode from protected mode, it cannot load the first stage, which runs in real mode. (These systems can load the first stage in real mode only.)

The rest of this section gives instructions for using the fourth method.

Although bootstrap loading is usually performed in response to an external event (such as a system reset or a monitor command), it can be initiated by an executing program. Such a program can load another system by calling the PUBLIC symbol BOOTSTRAP_ENTRY. To prepare for such a call, do the following:

1. Define BOOTSTRAP_ENTRY as an EXTERNAL symbol in the code of the invoking program.
2. Place a call to BOOTSTRAP_ENTRY in the code of the invoking program. The form of the call is

```
CALL BOOTSTRAP_ENTRY(@filename)
```

where:

filename	An ASCII string containing either the pathname of the target file followed by a CARRIAGE RETURN, or a CARRIAGE RETURN only. If the string contains a pathname, the named file is loaded. If the string contains a CARRIAGE RETURN only, the default file, as defined by the %DEFAULTFILE macro in the BS1.A86 or BS1MB2.A86 configuration file, is loaded. (The BS1.A86 and BS1MB2.A86 files are discussed in Chapter 3.)

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.)

3. Link the calling program to a version of the first stage of the Bootstrap Loader. You can do this by using the BS1.CSD file as a model and making the following changes:
   - Add the calling program to the list of modules that are linked in BS1.CSD.
   - "Comment out" the locate sequence if you want to use any code other than absolute code. For more details on absolute code, refer to the *iAPX 86, 88 Family Utilities Guide*.

   More information on the BS1.CSD file is available in Chapter 3.

2.4 CHOOSING A THIRD STAGE

If you plan to load iRMX II applications, you must include a version of the Bootstrap Loader's third stage on the secondary storage device from which you are loading your application. You can use the following kinds of third stages, depending on the type of system you are loading.

- A generic third stage
- A default device-specific third stage
- Your own configuration of the device-specific third stage containing customized device drivers.

The rest of this section should help you decide which third stage best suits your needs.

The important factors to consider when choosing a third stage are the size of your system, the type of mass storage devices you are using to boot your system, and the CPU board you are using.

If you plan to load your system from any of the Intel-supplied devices, you can use the default device-specific third stage regardless of the size of your system or a default generic third stage for systems up to 840K bytes. Both third stages are supplied for 80286 and 80386 CPU boards.

If you plan to load your system from a custom device, the size of the system determines which third stage you should use.

- For systems that are not expected to exceed 840K bytes, use the generic third stage. In this case, you do not need to supply a custom device driver for the third stage. You will already be supplying a custom first stage driver; the generic third stage will use that same driver to access the custom device.
- If your application exceeds 840K bytes, you must use the device-specific third stage, because it switches the processor into protected mode before loading the application. This enables the third stage to load into the entire 16 megabyte address space supported by protected mode. However, to load applications from your custom device, you must write a third stage device driver for your device. This driver can be a modification of your first stage driver that runs in the 80286 processor's protected mode. For information on writing a third stage driver, refer to Chapter 6.

NOTE

The 840K byte limit on systems loaded by the generic third stage applies to the boot file only. Once the boot file is loaded and has control, the entire 16 megabytes of memory address space is available for the system (both the free space manager and the Application Loader).

Table 2-1 lists the versions of the third stage that are supplied on the Bootstrap Loader Release Diskette. This table enables you to pick the appropriate third stage for your system. After you install your system, these files are available in the /RMX286/BOOT directory.

Table 2-1. Supplied Third Stage Files

CPU Board	Device-Specific Third Stage	Generic Third Stage
iSBC 286/10	28612	28612.GEN
iSBC 286/10A	28612	28612.GEN
iSBC 286/12	28612	28612.GEN
iSBC 286/100A	286100A	286100A.GEN
iSBC 386/2X	38620	38620.GEN
iSBC 386/3X	38620	38620.GEN
iSBC 386/116	386100	386100.GEN
iSBC 386/120	386100	386100.GEN

CHAPTER 3
CONFIGURING THE FIRST STAGE

3.1 INTRODUCTION

There are three stages to the Bootstrap Loader, and two of these stages (the first stage and the third stage) can be configured to match your application system. The second stage is constant and does not need to be configured. This chapter describes how to configure the first stage.

Configuring the first stage of the Bootstrap Loader involves the following operations:

- Editing three or more assembly language source files to indicate the configurable options and device drivers to include in the first stage.
- Invoking a SUBMIT file to assemble the source files, link them together with the code for the first stage, and assign absolute addresses to the code in preparation for placing it into PROM devices.

Default versions of the assembly language source files and the SUBMIT file are placed in the /RMX86/BOOT or /RMX286/BOOT directory during installation. These files include the following:

BS1.A86	This assembly language source file contains macros that specify information about the processor and the bus, how the boot device and load file are selected, and which devices can be booted from. You should use this file if your system is a MULTIBUS I system.
BS1MB2.A86	This assembly language source file contains macros that specify information about the processor and the bus, how the boot device and load file are selected, and which devices can be booted from. You should use this file if your system is a MULTIBUS II system.
BSERR.A86	This assembly language source file contains macros that tell the Bootstrap Loader what to do if errors occur during bootstrap loading.

B208.A86 BMSC.A86 B218A.A86 B224A.A86 B251.A86 B254.A86 B264.A86 BSCSI.A86	These assembly language source files contain configuration information about the first stage device drivers. Each file describes one device driver. For each device driver that you want to include in the first stage, you must set up the appropriate file and link it to the rest of the first stage.
BS1.CSD	This SUBMIT file contains the commands needed to assemble the preceding source files, link the resulting modules (and any others that you supply), and locate the resulting object module containing the configured first stage.

As shipped on the release diskettes, these files are set up to generate the default version of the Bootstrap Loader's first stage. If you decide to configure your own version of the first stage, you will most likely edit either the BS1.A86 or BS1MB2.A86 configuration file (depending upon your system), the BSERR.A86 configuration file, and the BS1.CSD submit file. Make changes in the device driver configuration files only if you want to change the Intel-supplied defaults in those files.

The following sections describe how to modify all the configuration files to tailor the first stage of the Bootstrap Loader to meet your specifications.

NOTE

It's important that the BS1.A86 or BS1MB2.A86 configuration file and the BS1.CSD SUBMIT file agree as to the device drivers that are included in the first stage. Whenever you change the device driver specifications in one of these files, be sure to check the other file as well. Specific areas that you should check are discussed in descriptions of the files.

3.2 BS1.A86 AND BS1MB2.A86 CONFIGURATION FILES

Figures 3-1 and 3-2 show the BS1.A86 and BS1MB2.A86 files as they are delivered from Intel. These files consist of four INCLUDE statements and several macros. The definitions of the macros that can appear in these files are contained in the INCLUDE file BS1.INC. The macros themselves are discussed in the next few sections.

NOTE

Depending on your system, you must choose between BS1.A86 and BS1MB2.A86 as the correct configuration file. If your system is a MULTIBUS I system, choose the BS1.A86 configuration file. If your system is a MULTIBUS II system, choose the BS1MB2.A86 configuration file.

```
    name    bsl

$include(:f1:bcico.inc)
$include(:f1:bmb2.inc)
$include(:f1:bmps.inc)
$include(:f1:bsl.inc)

%cpu(80386)


;   iSBC 188/48 initialization of the iAPX 188
;iAPX_186_INIT(y,0fc38h,none,80bbh,none,003bh)

;   iSBC 186/03(A) and iSBC 186/51 initialization of the iAPX 186
;iAPX_186_INIT(y,none,none,80bbh,none,0038h)

%console
%manual
%auto

%loadfile

%defaultfile('/system/rmx86')

%retries(5)

;clear_sdm_extensions

;cico

;  iSBC 86/05/12a/14/30/35
;serial_channel(8251a,0d8h,2,8253,0d0h,2,2,8)

;   iSBX 351 (on iSBX #0)
;serial_channel(8251a,0A0h,2,8253,0B0h,2,2,8)
```

Figure 3-1. Intel-Supplied BS1.A86 File

```
;   iSBX 354 Channel A (on iSBX #0)
;serial_channel(82530,084H,2,82530,084H,2,0,0eh,a)

;   iSBX 354 Channel B (on iSBX #0)
;serial_channel(82530,080H,2,82530,080H,2,0,0eh,b)

;   8 MHz iSBC 186/03A Channel A
;serial_channel(8274,0d8h,2,80186,0ff00h,2,0,0dh)

;   8 MHz iSBC 186/03A Channel B
;serial_channel(8274,0dah,2,80186,0ff00h,2,1,0dh)
;serial_channel(8274,0dah,2,80130,0e0h,2,2,034h)

;   6 MHz iSBC 186/03/51 Channel A
;serial_channel(8274,0d8h,2,80186,0ff00h,2,0,0ah)

;   6 MHz iSBC 186/03/51 Channel B
;serial_channel(8274,0dah,2,80186,0ff00h,2,1,0ah)
;serial_channel(8274,0dah,2,80130,0e0h,2,2,027h)

;   iSBC 188/48/56 SCC #1 Channel A
;serial_channel(82530,0d0h,1,82530,0d0h,1,0,0eh,a)

;   iSBC 188/48/56 SCC #1 Channel B
;serial_channel(82530,0d2h,1,82530,0d2h,1,0,0eh,b)

;   iSBC 286/10(A)/12 Channel A
;serial_channel(8274,0d8h,2,8254,0d0h,2,2,8)

;   iSBC 286/10(A)/12 Channel B
;serial_channel(8274,0dah,2,8254,0d0h,2,1,8)

;   iSBC 386/2X and iSBC 386/3X
;serial_channel(8251a,0d8h,2,8254,0d0h,2,2,8)
```

Figure 3-1. Intel-Supplied BS1.A86 File (continued)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                    Multibus I devices                                   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%device(af0, 0, deviceinit208gen, deviceread208gen)
%device(af1, 1, deviceinit208gen, deviceread208gen)
%device(w0, 0, deviceinitmscgen, devicereadmscgen)
%device(w1, 1, deviceinitmscgen, devicereadmscgen)
%device(wf0, 8, deviceinitmscgen, devicereadmscgen)
%device(wf1, 9, deviceinitmscgen, devicereadmscgen)
%device(s0, 0, deviceinitscsi, devicereadscsi)
%device(sx1410a0, 0, deviceinitscsi, devicereadscsi, sasi_x1410a)
%device(sx1410b0, 0, deviceinitscsi, devicereadscsi, sasi_x1410b)
%device(smf0, 2, deviceinitscsi, devicereadscsi, sasi_x1420mf)
%device(pmf0, 0, deviceinit218A, deviceread218A)
%device(pb0, 0, deviceinit251, deviceread251)
%device(b0, 0, deviceinit254, deviceread254)
%device(ba0, 0, deviceinit264, deviceread264)

%end
```

Figure 3-1. Intel-Supplied BS1.A86 File (continued)

```
    name    bsl

$include(:f1:bcico.inc)
$include(:f1:bmb2.inc)
$include(:f1:bmps.inc)

;bist(0FFFFH:0FFFFH)

;copy(08000H,00FFH,08000H,000FH,08000H,0H)

;   (LBX), (PSB,addr) or (LBX+PSB)
;auto_configure_memory(LBX)

$include(:f1:bsl.inc)

%cpu(80386)


;   MPC and ADMA configuration for iSBC 286/100 with iEXM 100 MPC module
;bmps(00H, 4, 08BH, 200H, 3, 2, 0A0H, 16)

;   MPC and ADMA configuration for iSBC 286/100A
;bmps(00H, 4, 08BH, 200H, 2, 3, 0E0H, 16)

;   MPC and ADMA configuration for iSBC 386/100
%bmps(00H, 4, 089H, 200H, 2, 3, 000H, 16)


%console
%manual
%auto

%loadfile

%defaultfile('/system/rmx86')

%retries(5)

;clear_sdm_extensions

;cico
```

Figure 3-2. Intel-Supplied BS1MB2.A86 File

```
;   iSBX 351 (on iSBX #0)
;serial_channel(8251a,0A0h,2,8253,0B0h,2,2,8)

;   iSBX 354 Channel A (on iSBX #0) for iSBC 386/100
;serial_channel(82530,084H,2,82530,084H,2,0,0eh,a)

;   iSBX 354 Channel B (on iSBX #0) for iSBC 386/100
;serial_channel(82530,080H,2,82530,080H,2,0,0eh,b)

;   iSBC 286/100A Channel A
;serial_channel(82530,0dch,2,82530,0dch,2,0,0eh,a)

;   iSBC 286/100A Channel B
;serial_channel(82530,0d8h,2,82530,0d8h,2,0,0eh,b)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                     Multibus II devices                               ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%device(s0, 0, deviceinitscsi, devicereadscsi)
%device(sx1410a0, 0, deviceinitscsi, devicereadscsi, sasi_x1410a)
%device(sx1410b0, 0, deviceinitscsi, devicereadscsi, sasi_x1410b)
%device(smf0, 2, deviceinitscsi, devicereadscsi, sasi_x1420mf)
%device(pmf0, 0, deviceinit218A, deviceread218A)
%device(w0, 0, device_init_224a, device_read_224a)
%device(w1, 1, device_init_224a, device_read_224a)
%device(wf0, 4, device_init_224a, device_read_224a)
%device(wf1, 5, device_init_224a, device_read_224a)
%end
```

Figure 3-2. Intel-Supplied BS1MB2.A86 File (continued)

To configure your own version of the Bootstrap Loader first stage, edit either the BS1.A86 or BS1MB2.A86 file if you need to include or exclude macros. A percent sign (%) preceding the macro name includes (invokes) the macro, and a semicolon (;) preceding the macro name excludes the macro, treating it as a comment.

NOTE

When you exclude a macro, you must replace the percent sign with a semicolon. Do not simply add a semicolon in front of the percent sign.

The order in which the macros should appear is the same order that they are listed in the BS1.A86 or BS1MB2.A86 file.

The following sections describe the macros that can appear in the BS1.A86 and BS1MB2.A86 files. Because the Bootstrap Loader supports both iRMX I and iRMX II Operating Systems, some of these macros apply to one Operating System and not the other. In such cases, the section heading notes the operating system to which the macro applies. When no operating system designation appears, the macro is valid for both the iRMX I and iRMX II Operating Systems. The macros are described in the order they are listed in the BS1.A86 and BS1MB2.A86 files.

If you make a syntax error when entering macros into the BS1.A86 or BS1MB2.A86 file, an error message appears when assembling the file. For example, if you misspell a macro name in a macro call, the following type of message may be returned:

```
*** ERROR #301 IN 129, (MACRO) UNDEFINED MACRO NAME
                                    INSIDE CALL: BADNAME
***_____________________________________↑
*** ERROR #1 IN 129, SYNTAX ERROR
```

If an error such as this occurs, check for correctness in the BS1.A86 or BS1MB2.A86 file and attempt to reassemble the file.

3.2.1 %BIST Macro (MULTIBUS® II Only)

MULTIBUS II systems include a Built-In Self Test (BIST) program in PROM devices that verifies MULTIBUS II hardware when the hardware is powered up. The %BIST macro causes the Bootstrap Loader to invoke the BIST program on the CPU board during Bootstrap Loader initialization. The BIST program then tests the hardware.

If the BIST program encounters an error condition, it places an error code in the AX register and loops. It does not call the Bootstrap Loader's BSERROR routine because an error of this type implies that the system hardware is inoperable.

The %BIST macro should be included only for MULTIBUS II systems, and only for those systems that don't also include a monitor in PROM devices. In systems that include a monitor, the monitor becomes active before the Bootstrap Loader, and it invokes the BIST program. Therefore, invoking the BIST program from the Bootstrap Loader is unnecessary.

The syntax of the macro is

%BIST (address)

where:

address	Address of the CPU board's BIST program. This parameter must be entered in the form BASE:OFFSET (for example, 1234:5678). To determine the address of your CPU board's BIST program, refer to the hardware reference manual for that board.

3.2.2. %COPY Macro (MULTIBUS® II Only)

The %COPY macro is used with 386/116- and 386/120-based systems. If your system is not of this type, do not include the %COPY macro in the BS1MB2.A86 file.

Both 386/116- and 386/120-based systems locate EPROM memory at the top of the 4 gigabyte address space supported by the 80386 upon reset. However, the first stage of the Bootstrap Loader must execute within the first megabyte of address space (real mode). Because the first stage must be repositioned within memory, you must use the %COPY macro for any application where the EPROM memory is mapped outside of the first megabyte of address space upon reset.

In contrast, the 386/2X and 386/3X systems locate EPROM memory at the top of the first megabyte of memory space upon reset. Thus, the %COPY macro is unnecessary.

This macro copies the first stage of the Bootstrap Loader from EPROM devices to the low megabyte of RAM. You should only specify this macro if you do not have a monitor installed and the Bootstrap loader executes first upon system reset.

The syntax of the %COPY macro is as follows:

%COPY(src_lo, src_hi, dest_lo, dest_hi, count_lo, count_hi)

where:

src_lo	The low word of the 24-bit physical source address.
src_hi	The high byte of the 24-bit physical source address.
dest_lo	The low word of the 24-bit physical destination address.
dest_hi	The high byte of the 24-bit physical destination address.
count_lo	The low word of the number of bytes in the first stage.
count_hi	The high byte of the number of bytes in the first stage.

3.2.3. %AUTO_CONFIGURE_MEMORY Macro (MULTIBUS® II Only)

This macro causes the Bootstrap Loader to automatically configure the starting and ending addresses of all iLBX and/or iPSB memory boards available to MULTIBUS II systems. Configuration begins with the memory board in the lowest numbered slot and progresses through the memory board in the highest numbered slot. Configuration occurs differently depending upon how you invoke the macro.

You should include the %AUTO_CONFIGURE_MEMORY macro only for MULTIBUS II systems, and only for those systems in which the Bootstrap Loader is invoked upon system reset (as opposed to under program control). In systems that include the monitor in PROM devices, the monitor becomes active before the Bootstrap Loader, and it should invoke its own %AUTO_CONFIGURE_MEMORY macro. Therefore, invoking the macro from the Bootstrap Loader is unnecessary.

The syntax of the macro is

%AUTO_CONFIGURE_MEMORY(interface_type [,start_address])

where:

interface_type	A string representing the bus interface of the memory board(s) to be configured. Valid strings are LBX, PSB, or LBX+PSB.
start_address	The starting 64K page of memory when PSB memory is being configured.

Three possible configuration options exist: iLBX only, iPSB only, or iLBX and iPSB. You must specify the required parameters using one of the following three methods:

%AUTO_CONFIGURE_MEMORY (LBX)

This option configures memory boards accessible to the processor via the iLBX bus. Using this configuration option, the macro assigns sequential consecutive addresses beginning with zero for the start and stop addresses of each iLBX memory board. Board configuration proceeds from the board occupying the lowest slot number to the board occupying the highest slot number.

%AUTO_CONFIGURE_MEMORY (PSB, start address)

This option configures memory boards accessible to the CPU via the iPSB bus. Using this configuration option, the macro assigns sequential consecutive addresses for the start and stop addresses of each iPSB memory board. The assigned addresses begin with the supplied starting address. Board configuration proceeds from the board occupying the lowest slot number to the board occupying the highest slot number.

%AUTO_CONFIGURE_MEMORY (LBX+PSB)

This option configures memory in the same manner as the first option, with one additional configuration. All boards on the iLBX bus that also have iPSB interfaces have the same starting and ending addresses for both interfaces.

The following syntax errors can occur if you enter incorrect parameters or incorrect combinations of parameters.

```
ERROR - <type>, invalid interface type
ERROR - invalid parameter combination
```

3.2.4 %CPU Macro

The %CPU macro identifies the type of CPU that performs the bootstrap loading operation. You must include this macro in the BS1.A86 or BS1MB2.A86 file once (and only once).

The syntax of the CPU macro is

%CPU(cpu_type)

where:

cpu_type The type of CPU performing the bootstrap operation. Valid types are:

Type	Description
8086	8086 processor (iRMX I only)
8088	8088 processor (iRMX I only)
80186	80186 processor (iRMX I only)
80188	80188 processor (iRMX I only)
80286	80286 processor (iRMX I and iRMX II)
80386	80386 processor (iRMX I and iRMX II)

3.2.5 %BMPS Macro (MULTIBUS® II Only)

The %BMPS macro configures the message passing system used during bootstrap loading. This macro identifies the base address of the Message Passing Coprocessor (MPC), address distance between MPC ports, and information that defines how direct memory access (DMA) transfers occur. If you have a MULTIBUS II system that bootloads from a device whose driver uses MULTIBUS II transport protocol (i.e. the 186/224A driver), you must use this macro. If you have a MULTIBUS I system or a system that bootloads from a device whose driver does not use MULTIBUS II transport protocol, you must not use this macro.

The syntax of the %BMPS macro is

```
%BMPS (mpc$base$addr, port$sep, duty$cycle, dma$base$addr, dma$in, dma$out,
       dma$trans, data$width)
```

where:

mpc$base$addr	The base I/O port address of the MPC. Refer to the appropriate single board computer user's guide for this address.
port$sep	The number of addresses separating individual MPC ports. For example, if the mpc$base$addr is 0000H and the next three I/O port addresses are 0004H, 0008H, and 000CH, respectively, the port$sep is 4H. Refer to the appropriate single board computer user's guide for the I/O port address map.
duty$cycle	The MPC duty cycle for the local bus. (The rate at which data packets are generated.) For information on how to calculate a duty cycle suitable for the local bus, refer to the *MPC User's Manual*. For duty cycles suitable for Intel single board computers, refer to the appropriate single board computer user's guide.
dma$base$addr	The base I/O port address for the Advanced Direct Memory Access (ADMA) controller. Refer to the appropriate single board computer user's guide for this address.
dma$in	The channel used to receive (input) DMA message passing transfers. Refer to the appropriate single board computer user's guide for this channel number.
dma$out	The channel used to send (output) DMA message passing. Refer to the appropriate single board computer user's guide for this channel number.
dma$trans	The I/O port address used for DMA data transfers. Refer to the appropriate single board computer user's guide for this address.
data$width	The data width in bits of the local bus. This value must be either 16 or 32 (decimal). If the width is set to 32 bits on a 386/116- or 386/120-based board, flyby (one cycle) DMA mode is enabled.

The %BMPS macro can generate errors if the local bus width is not 16 or 32 bits wide.

3.2.6 %iAPX_186_INIT Macro (iRMX I MULTIBUS® I Systems Only)

The %iAPX_186_INIT macro specifies the initial chip select and mode values for 80186 and 80188 CPUs. Include this macro only for systems that use the 80186 or 80188 processor and do not include a monitor in PROM devices. In systems that include the iSDM monitor, the monitor becomes active before the Bootstrap Loader, and the monitor must initialize the CPU. An iSDM configuration macro is available for this purpose. See the *iSDM System Debug Monitor Reference Manual* for more information.

The syntax of the iAPX_186_INIT macro is

```
%iAPX_186_INIT(rmx, umcs, lmcs, mmcs, mpcs, pacs)
```

where:

rmx — The initial mode of the 80186 Programmable Interrupt Controller (PIC). Acceptable values are as follows:

Value	Description
y	The 80186 PIC is initialized in iRMX compatibility mode.
n	The 80186 PIC is initialized in default mode.

umcs — Initial value for the upper-memory chip-select control register.

lmcs — Initial value for the lower-memory chip-select control register.

mmcs — Initial value for the midrange-memory chip-select control register.

mpcs — Initial value for the memory-peripheral chip-select control register.

pacs — Initial value for the peripheral-address chip-select control register.

In all parameters except the first one (rmx), NONE is also an acceptable value, implying that no initialization value should be placed in the corresponding register. For information on the chip-select control registers, and the values to place in them, see the data sheets for the 80186 and 80188 processors.

All the default parameter values for this macro (in the Intel-supplied BS1.A86 file shown in Figure 3-1) are appropriate to initialize the CPUs on the iSBC 186/03(A), iSBC 186/51 and iSBC 188/48/56 boards.

The iRMX I Operating System does not allow you to move the 80186 relocation register to I/O addresses other than 0FF00H, its default register.

3.2.7 %CONSOLE, %MANUAL, and %AUTO Macros

The CONSOLE, MANUAL, and AUTO macros specify how the first stage identifies the file that the second stage will load (either the load file or the third stage) and the device on which the file is found.

The syntax of the %CONSOLE, %MANUAL, and %AUTO macros is

%CONSOLE

%MANUAL

%AUTO

There are no parameters associated with any of these macros.

Depending on the action you want the Bootstrap Loader to take, you can include none, any, or all of these macros, and the combination you choose defines the set of actions taken. Because the %MANUAL macro automatically includes both the %CONSOLE and %AUTO macros, five functionally-distinct combinations are possible. Each of these combinations requires that the device list at the end of the BS1.A86 or BS1MB2.A86 file be set up in a certain way. For more information on the device list, see the discussion of the %DEVICE macro later in this chapter. The following paragraphs list the possible macro combinations, the device requirements, and the actions that the Bootstrap Loader takes when each combination is invoked.

No %CONSOLE, %MANUAL, or %AUTO macro

(Requires that the device list defined with %DEVICE macros have only one entry.)

- The Bootstrap Loader tries once to load from the active device.
- The Bootstrap Loader tries once to load the file with the default pathname (the one you define with the %DEFAULTFILE macro).

%CONSOLE only (Requires that the device list have only one entry.)

- The Bootstrap Loader tries once to load from the device in the device list.
- The Bootstrap Loader issues an asterisk (*) prompt at the console terminal and waits for an operator to enter the pathname of the file to load. It tries once to load the file the operator specifies.
  - -- If the operator enters a pathname, the Bootstrap Loader loads the file with that pathname.
  - -- If the operator enters a CARRIAGE RETURN only, the file with the default pathname is loaded.

%MANUAL only (Requires a device list with at least one entry.)

- The Bootstrap Loader issues an asterisk (*) prompt for a pathname at the console terminal.
- The Bootstrap Loader chooses a device depending on the operator's response.
  - -- If a device name is entered, the Bootstrap Loader loads from that device. It tries to load until the device becomes ready or until no more tries are allowed (as limited by the optional %RETRIES macro).
  - -- If only CARRIAGE RETURN is entered, the Bootstrap Loader looks for a ready device by searching through the list of devices (in the order the %DEVICE macros are listed in the BS1.A86 or BS1MB2.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 the Bootstrap Loader finds a ready device, it loads from that device.
- The Bootstrap Loader chooses a file depending on the operator's response.
  - -- 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 (Requires a device list with at least one entry.)

- The Bootstrap Loader looks for a ready device by searching through the list of devices (in the order the %DEVICE macros are listed in the BS1.A86 or BS1MB2.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 the Bootstrap Loader finds a ready device, it tries once to load the file with the default file name.

%AUTO, %MANUAL, and %CONSOLE (Requires a device list with at least one entry.)

- The Bootstrap Loader issues an asterisk (*) prompt for a pathname at the console.
- If the operator responds with a pathname that contains no device name, the Bootstrap Loader looks for a ready device by searching through the list of devices (in the order the %DEVICE macros are listed in the BS1.A86 or BS1MB2.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 the Bootstrap Loader finds a ready device or the operator responds with a pathname containing a device name, the Bootstrap Loader 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 CARRIAGE RETURN is entered, it tries to load the file with the default pathname.

Whenever the Bootstrap Loader's asterisk prompt appears, the operator can include a Debug Switch along with a device and/or filename specification. The Debug Switch is described in Chapter 2.

3.2.8 %LOADFILE Macro

The %LOADFILE macro causes the Bootstrap Loader to display the pathname of the file it loads. If you are loading an iRMX I system, this will be the pathname of the load file. If you are loading an iRMX II system, the pathname of the Bootstrap Loader's third stage will be displayed. The macro displays the pathname at the console after loading the second stage but before loading the load file (or third stage).

If you include the %LOADFILE macro, you must also include either the %CONSOLE or %MANUAL macros to enable the Bootstrap Loader to access the console.

The syntax of the %LOADFILE macro is

```
%LOADFILE
```

There are no parameters associated with this macro.

3.2.9 %DEFAULTFILE Macro

The %DEFAULTFILE macro specifies the complete pathname of the default file. The default file is the file that the second stage loads whenever no other file is specified.

The syntax of the %DEFAULTFILE macro is

```
%DEFAULTFILE('pathname')
```

where:

pathname	Hierarchical pathname of the default file, starting at the root directory. The pathname must be enclosed in single quotes. For example, the name '/BOOT/RMX28612' might be used.

If you omit this macro from the BS1.A86 or BS1MB2.A86 file, a NULL pathname is assumed by the Bootstrap Loader first stage. In this case, the second stage assumes the default name is /SYSTEM/RMX86. The Intel-supplied BS1.A86 and BS1MB2.A86 files include a %DEFAULTFILE macro and assigns /SYSTEM/RMX86 as the default file.

3.2.10 %RETRIES Macro

The %RETRIES macro, when included with the %AUTO or %MANUAL macros, limits the number of times that the first stage searches the device list for a ready device.

NOTE

If you omit the %RETRIES macro when including the %AUTO or %MANUAL macros and no device in the list is ready, then the search for a ready device continues indefinitely.

The syntax of the %RETRIES macro is

```
%RETRIES(number)
```

where:

number — Maximum number of times the first stage checks each device for a ready condition. You can specify any number in the range of 1 through 0FFEH.

3.2.11 %CLEAR_SDM_EXTENSIONS Macro

The %CLEAR_SDM_EXTENSIONS macro causes the Bootstrap Loader to clear the iSDM monitor command extensions (the U, V, and W commands). Once cleared, a monitor extension, such as the iRMX I or iRMX II System Debugger (SDB) or the System 300 System Confidence Test (SCT), must be reinitialized before it can be used again.

This macro is useful when adding monitor-level debugging command extensions. It prevents you from inadvertently attempting to invoke a monitor extension that was loaded in a previous debugging session and overwriting application or Operating System code.

The syntax of this macro is

```
%CLEAR_SDM_EXTENSIONS
```

The Intel-supplied versions of the BS1.A86 and BS1MB2.A86 files do not invoke this macro. This macro must not be invoked if you are configuring a standalone Bootstrap Loader.

3.2.12 %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 the monitor. If you include the CICO macro, you must perform some other operations as well, depending on 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:

1. Change the line in the BS1.CSD file (Figure 3-3) that reads

   ```
   &    :f1:bcico.obj, &
   ```

   to

   ```
   :f1:bcico.obj, &
   ```

2. Include exactly one instance of the %SERIAL_CHANNEL macro (described in the next section) in the BS1.A86 or BS1MB2.A86 file.

If you supply your own standalone CI and CO routines:

1. Change the line in the BS1.CSD file (Figure 3-3) that reads

```
&    :f1:bcico.obj, &
```

to

```
:f1:mycico.obj, &
```

where:

mycico.obj	An object file that you supply containing procedures named CI, CO, and CINIT. CINIT must perform initialization functions required to prepare the console for input and output operations.

2. Do not include the %SERIAL_CHANNEL macro in the BS1.A86 or BS1MB2.A86 file.

The syntax of the %CICO macro is

```
%CICO
```

There are no parameters associated with this macro. The CICO macro is not invoked in the Intel-supplied BS1.A86 or BS1MB2.A86 file. This macro must be invoked if you are configuring a standalone Bootstrap Loader which prompts for the load file pathname.

3.2.13 %SERIAL_CHANNEL Macro

The %SERIAL_CHANNEL macro identifies the type and characteristics of the serial channel used to communicate with your system console.

You must omit this macro if any of the following conditions are true:

- Your system includes a monitor.
- Your system does not use a terminal during bootstrap loading.
- You supply your own CI and CO routines.

NOTE

You cannot use the %SERIAL_CHANNEL macro unless the serial device is local to the CPU board. Also, the %SERIAL_CHANNEL macro does not support the on-board diagnostic serial port on the iSBC 386/100 board.

You must include this macro if you are configuring a standalone Bootstrap Loader to use the Intel-supplied CI and CO routines (see the description of the %CICO macro in the previous section). In this case, use the %SERIAL_CHANNEL macro to describe the serial controller device that handles the communication to and from the terminal accessed by the Bootstrap Loader.

The Bootstrap Loader permits serial communication via an 8251A USART, an 8274 Multi-Protocol Serial Controller, or an 82530 Serial Communications Controller. The Intel-supplied BS1.A86 and BS1MB2.A86 files list appropriate invocations of the %SERIAL_CHANNEL macro for each of these serial channel controllers. To choose one of these versions of the macro, replace the semicolon on the appropriate line with a percent sign. Including more than one %SERIAL_CHANNEL macro causes an assembly error in BS1.A86 or BS1MB2.A86.

The syntax 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. Valid values are 8251A, 8274, and 82530.

serial_base_port — The 16-bit port address of the base port used by the serial channel. This port varies according to the type of serial controller device and, if applicable, the channel used on the device. To determine the port whose address you should specify here, look at the left column of the following list. Pick the item that corresponds to the serial device on your CPU board and the channel through which the CPU communicates with your terminal. Then specify the port address of the corresponding port listed in the right column. The hardware reference manual for your CPU board lists the port addresses for these serial devices.

Serial Channel	Base Port
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 separating 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. Valid values are:

8253	8254
80130	80186
82530	NONE

Specifying NONE implies that the baud rate timer is automatically initialized and the Bootstrap Loader does not need to perform this function.

counter_base_port

The 16-bit port address of the base port used by the baud rate timer. The port whose address you specify varies according to the type of timer device and, if applicable, the channel used on the device. The following list shows the ports for each of the valid timers. Specify the address of the port that corresponds to your timer device. The hardware reference manual for the CPU board lists the port addresses for these serial devices.

Timer Type	Base Port
8253	Counter 0 Count Register Port
8254	Counter 0 Count Register Port
80130	ICW1 Register Port
80186	Use 0FF00H for all boards
82530 Channel A	Channel A Command Register Port
82530 Channel B	Channel B Command Register Port

counter_port_delta

The number of bytes separating consecutive ports used by the timer.

baud_counter

The number of the counter that is used for baud-rate generation. The following list identifies the possible counter numbers you can specify for each of timers.

Timer Type	Counter Numbers
8253	0, 1, or 2
8254	0, 1, or 2
80130	2
80186	0 or 1
82530	0

count — A value that when loaded into the timer register generates the desired baud rate. The method of calculating this value follows these parameter definitions.

flags — Applies only when the serial type parameter is defined as 82530. For other serial controllers, omit this parameter.

This parameter specifies which channel of an 82530 Serial Communications Controller will serve as the serial controller. Valid values are

Value	Channel
A	Channel A
B	Channel B

To derive the correct value for the count parameter, you must perform five computations. The starting values for these computations are the desired baud rate at which you want the serial port to operate and the clock input frequency to the timer. The clock input frequency is listed in the data sheet for the timer.

First, perform one of the following calculations to obtain a temporary value for use in later calculations:

If the timer is an 8253, 8254, 80130, or 80186,

temporary_value = (clock frequency in Hz)/(baud rate x 16)

If the timer is an 82530,

temporary_value = ((clock frequency in Hz)/(baud rate x 2)) - 2

Next, perform the following calculation to obtain the fractional part of the temporary value found in the first calculation:

fraction = temporary_value - INT(temporary_value)

The INT function gives the integer portion of temporary_value.

The third and fourth calculations yield the desired count value and another value, called error_fraction. The error_fraction value is needed to determine whether the calculated count value is feasible, given the clock frequency specified in the first calculation. These calculations, performed according to the size of the value of "fraction" from the second calculation, are as follows:

If the value of "fraction" is greater than or equal to .5,

count = INT (temporary_value) + 1
error_fraction = 1 - fraction

If the value of "fraction" is less than .5,

count = INT (temporary_value)
error_fraction = fraction

The fifth and final calculation yields the percentage of error that occurs when the clock frequency is used to generate the baud rate, as follows:

% error = (error_fraction / count) x 100

If the % error value is less than 3, then the calculated count value is appropriate, and the desired baud rate will be generated by the specified clock frequency. However, if the % error value is 3 or greater, you must do one or both of the following:

- 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 to get a new "count" value and to see whether the revised value of "% error" is less than 3. Continue this process until the "% error" value is less than 3.

The %SERIAL CHANNEL macro can generate the following error messages:

```
ERROR - invalid port delta for the (ser_type) Serial Device
ERROR - <ser_type> is an invalid Serial Channel type
ERROR - Invalid port delta for the Baud Rate Timer
ERROR - 8253/4 Baud Rate Counter is not 0, 1 or 2
ERROR - Counter 2 is the only valid 80130 Baud Rate Counter
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 - Max Baud Rate Count must be greater than 1
```

3.2.14 %DEVICE Macro

The %DEVICE macro defines a device unit from which your application system can be bootstrap loaded. If the BS1.A86 or BS1MB2.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.

All %DEVICE macros that select device units on the same controller must be listed consecutively in BS1.A86 or BS1MB2.A86, or assembly errors will occur. Recall that multiple %DEVICE macros may be included only if the %AUTO or %MANUAL macro is included (otherwise, an error occurs during the assembly of BS1.A86 or BS1MB2.A86).

The syntax of the %DEVICE macro is

```
%DEVICE(name, unit, device$init, device$read, unit_info)
```

where:

name — The physical name of the device, not enclosed in quotes or between colons. This is the name that you would enter to specify this device when invoking the Bootstrap Loader from the keyboard. (However, when invoking the Bootstrap Loader, you would surround this name with colons.)

After the Bootstrap Loader loads from a device, it passes the physical name of the device, as listed here, to the load file. To enable the Operating System's Automatic Boot Device Recognition capability (see Appendix A) to function, this physical name must match a device-unit name for the device as specified during the configuration of the Operating System. Refer to the *Interactive Configuration Utility Reference Manual* for more information about configuring the Operating System.

unit — The number of this unit on this device. Unit numbering is the same as that used for devices by the Basic I/O System. Refer to the *Device Driver User's Guide* for more information about unit numbering.

device$init — The name of the device initialization procedure that is part of the first stage device driver for this device-unit. Before attempting to read from the device-unit, the Bootstrap Loader calls this procedure to perform initialization functions. If the device-unit has an Intel-supplied device driver, specify the name of the device initialization procedure as listed in Table 3-1. If you supply your own driver (written as described in Chapter 5), enter the name of the initialization procedure.

device$read — The name of the device read procedure that is part of the first stage device driver for this device-unit. To read from this device-unit, the first and second stages of the Bootstrap Loader call this procedure. If your Bootstrap Loader uses a generic third stage, it too uses this device read procedure to read from the device unit. If the device-unit has an Intel-supplied device driver, specify the name of the device read procedure as listed in Table 3-1. If you supply your own driver (written as described in Chapter 5), enter the name of the device read procedure.

unit_info — An ASM86 label that marks the location of an array of BYTEs containing specific device-unit information required by the mass storage device defined by this invocation of the %DEVICE macro.

This parameter is currently used only by the SCSI device driver. If you include it for any other device, the Bootstrap Loader will fail to load your application from that device. Refer to the "First Stage Device Driver Files" section of this chapter, under the descriptions of the %SCSI and %SASI_UNIT_INFO macros for information about how and when to specify this unit information and for examples of its use.

Table 3-1 lists the names of the device initialization and device read procedures for Intel-supplied first stage device drivers.

Table 3-1. Procedure Names for Intel-Supplied First Stage Drivers

Device Driver	Device Initialize Procedure	Device Read Procedure
iSBC 208 Specific Driver	deviceinit208	deviceread208
iSBC 208 General Driver	deviceinit208gen	deviceread208gen
iSBC MSC Specific Driver*	deviceinitmsc	devicereadmsc
iSBC MSC General Driver*	deviceinitmscgen	devicereadmscgen
SCSI Driver	deviceinitscsi	devicereadscsi
iSBX 218A Driver	deviceinit218A	deviceread218A
iSBC 224A Driver	deviceinit224A	deviceread224A
iSBC 251 Driver	deviceinit251	deviceread251
iSBC 254 Driver	deviceinit254	deviceread254
iSBC 264 Driver	deviceinit264	deviceread264

* The MSC drivers support the iSBC 214, iSBC 215G, iSBC 220 controllers, as well as the iSBX 218A controller mounted on the iSBC 215G board. The drivers must be reconfigured to support the iSBC 220 controller.

Table 3-1 lists both specific and general procedures for the iSBC 208 and MSC devices. Configurations of the Bootstrap Loader that use the general version of either driver will be larger.

One difference between the two versions of these device drivers is that the general versions will bootstrap load applications from any of the standard types of diskettes as defined in the Installation Systems. The specific versions will bootstrap load applications only from specific types of diskettes listed in Tables 3-2 and 3-3. These tables apply to the specific versions of both the iSBC 208 and MSC device drivers.

Table 3-2. 5.25-Inch Diskettes Supported by iSBC 208 and MSC-Specific Drivers

Sector Size	Density	Sectors per Track
256	Single	9
256	Double	16
NOTE: The diskettes can be formatted with either 48 tracks per inch or 96 tracks per inch, and can be either single- or double-sided.		

Table 3-3. 8-Inch Diskettes Supported by iSBC 208 and MSC-Specific Drivers

Sector Size	Density	Sectors per Track
128	Single	26
256	Double	26
NOTE: The diskettes may be either single- or double-sided.		

The Intel-supplied BS1.A86 and BS1MB2.A86 configuration files include %DEVICE macros for all of the supported devices, and include multiple instances of some of the macros to indicate multiple units on the same device. It doesn't hurt to include support for all of these devices, even if your application system won't contain all of them. And if you add a new device later, you'll be able to boot from the device without generating new boot PROM devices. However, you can reduce the size of your Bootstrap Loader by excluding support for devices that you never intend to use. Release 3.2 of the iSDM monitor provides space from 0FE400H to 0FFF7FH for use by the Bootstrap Loader. This requires you choose only the devices you need when you reconfigure the Bootstrap Loader so it will fit into the space allocated by the iSDM monitor. If the Bootstrap Loader does not fit into the space allocated by the monitor, you must locate it below the monitor.

To exclude a device driver from the Bootstrap Loader, two steps must be performed. First, exclude all the %DEVICE macros in BS1.A86 or BS1MB2.A86 that apply to device units on that controller. To do this, edit BS1.A86 or BS1MB2.A86 and replace the percent sign (%) in front of the macro with a semicolon (;). The edited version of such a macro would look similar to:

```
;device(ba0, 0, deviceinit264, deviceread264)
```

The semicolon replacing the percent sign turns the %DEVICE macro for the iSBC 264 driver (in this case) into a comment.

Second, edit the file BS1.CSD as described later in this chapter.

3.2.15 %END Macro

The %END macro is required at the end of the BS1.A86 or BS1MB2.A86 file. The syntax of this macro is

```
%END
```

There are no parameters associated with the %END macro.

3.3 BSERR.A86 CONFIGURATION FILE

The BSERR.A86 file, shown in Figure 3-3, 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
;int1
%int3
;halt

%end
```

Figure 3-3. 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 (;) replaces the percent sign, 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 four macros, %AGAIN, %INT1, %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.

3.3.1 %CONSOLE Macro

The %CONSOLE macro causes the Bootstrap Loader to display a brief message at the console whenever a bootstrap loading error occurs. The message indicates the nature of the error (see Chapter 7 for the message list).

The syntax of the %CONSOLE macro is

```
%CONSOLE
```

There are no parameters associated with this macro.

This %CONSOLE macro is completely unrelated to the %CONSOLE macro used in the BS1.A86 or BS1MB2.A86 file. Be careful not to confuse them.

3.3.2 %TEXT Macro

The %TEXT macro is similar to 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 makes the first stage of the Bootstrap Loader larger.

The syntax of the %TEXT macro is

```
%TEXT
```

There are no parameters associated with this macro. If you include the %TEXT macro, the %CONSOLE macro is automatically included.

3.3.3 %LIST Macro

The %LIST macro causes the Bootstrap Loader to display a list of the ready device-units at the console whenever the operator enters an invalid device-unit name. You can include this macro only if you include the %MANUAL macro in the BS1.A86 or BS1MB2.A86 file, as described earlier in this chapter.

The syntax of the %LIST macro is

```
%LIST
```

There are no parameters associated with this macro. If you include the %LIST macro, the %CONSOLE and %TEXT macros are automatically included.

3.3.4 %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. You should 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 syntax of the %AGAIN macro is

```
%AGAIN
```

Exactly one of the %AGAIN, %INT1, %INT3, and %HALT macros must be included, or an error will occur when BSERR.A86 is assembled.

3.3.5 %INT1 Macro

The %INT1 macro causes the Bootstrap Loader to execute an INT 1 (software interrupt) instruction whenever a bootstrap loading error occurs. This macro useful for passing control to the D-MON386 monitor. The iSDM monitor does not support this macro.

The syntax of the %INT1 macro is

```
%INT1
```

There are no parameters associated with this macro.

Exactly one of the %AGAIN, %INT1, %INT3, and %HALT macros must be included, or an error will occur when BSERR.A86 is assembled.

The %INT1 macro, as well as the %INT3 and %HALT macros described next, are reasonable choices if none of the %CONSOLE, %TEXT, or %LIST macros are included in the BSERR.A86 file.

3.3.6 %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 monitor, the INT 3 instruction passes control to the monitor. Otherwise, the INT 3 instruction has no effect unless you have placed the address of your custom interrupt handler in position 3 of the interrupt vector table.

The syntax of the %INT3 macro is

```
%INT3
```

There are no parameters associated with this macro.

Exactly one of the %AGAIN, %INT1, %INT3, and %HALT macros must be included, or an error will occur when BSERR.A86 is assembled.

The %INT3 macro, as well as the %INT1 and %HALT macros, are reasonable choices if none of the %CONSOLE, %TEXT, or %LIST macros are included in the BSERR.A86 file.

3.3.7 %HALT Macro

The %HALT macro causes the Bootstrap Loader to execute a halt instruction whenever a bootstrap loading error occurs.

The syntax of the %HALT macro is

```
%HALT
```

There are no parameters associated with this macro.

Exactly one of the %AGAIN, %INT1, %INT3, and %HALT macros must be included, or an error will occur when BSERR.A86 is assembled.

The %HALT macro, as well as the %INT1 and %INT3 macros, are reasonable choices if none of the %CONSOLE, %TEXT, or %LIST macros are included in the BSERR.A86 file.

3.3.8 %END Macro

The %END macro is required at the end of the BSERR.A86 file.

The syntax of this macro is

```
%END
```

There are no parameters associated with the %END macro.

3.4 DEVICE DRIVER CONFIGURATION FILES

A separate configuration file is included for each device driver provided with the Bootstrap Loader. These files are named B208.A86, BMSC.A86, B218A.A86, B224A.A86, B251.A86, B254.A86, B264.A86, and BSCSI.A86. Each consists of an include statement and a macro call. The source file always has the form

```
$include(:f1:bxxx.inc)

%bxxx( parameters )
```

where:

xxx	Either 208, MSC, 218A, 224A, 251, 254, 264, or SCSI, depending on the device driver.

The number and type of parameters that are included with the macro depend on the device driver. The parameters for each macro are discussed in the following sections. Additionally, when a SASI controller board is used with the SCSI device driver, it requires another macro. Refer to the "%BSCSI Macro" and "%SASI_UNIT_INFO Macro" sections for details and for invocation examples. The default parameter values for the macros in these sections are compatible with the default parameter values of the Installation Systems.

You should prepare one of these files for each type of device you want the first stage of the Bootstrap Loader to support. In most cases, you can use the Intel-supplied files. The following sections describe the individual macros so that you can make changes to them, if necessary.

3.4.1 %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 invocation of this macro in the B208.A86 file is

%B208(180H)

3.4.2 %BMSC and %B220 Macros

The BMSC.A86 file contains two macros, %BMSC and %B220. However, you can use only one. If you have one of the drivers listed at the bottom of Table 3-1, you should use the %BMSC macro. If you have the iSBC 220, you should use the %B220 macro. Both macros have the form

%Bxxx (wakeup, cylinders, fixed_heads, removable_heads, sectors,
dev_gran, alternates)

where:

xxx Either MSC or 220.

wakeup Base address of the controller's wakeup port.

The remaining parameters are used to specify the characteristics of the disk drives. If the %DEVICE macro you used for MSC or iSBC 220 devices in the BS1.A86 or BS1MB2.A86 file has deviceinitmsc (rather than deviceinitmscgen) as its third parameter, then all MSC or iSBC 220 drives used by the Bootstrap Loader must have the characteristics listed in the following parameters. 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 deviceinitmscgen, these restrictions do not apply and the following parameters are not used by the Bootstrap Loader.

cylinders Number of cylinders on the disk drive or drives.

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 BMSC.A86 file, the default invocation of the %BMSC macro is

```
%BMSC(100H, 256, 2, 0, 9, 1024, 5)
```

and the default form of the uninvoked %B220 macro is

```
;B220(100H, 256, 2, 0, 9, 1024, 5)
```

3.4.3 %B218A Macro

The %B218A macro has the form

```
%B218A(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. Valid values are:

Value	Description
0FFH	The drive will be turned off after bootstrap loading. Specify this value only if this device is not to become the system device. Turning off the drive slows slows bootstrap loading.
00H	The drive will not be turned off after bootstrap loading.

The default invocation of this macro in the B218A.A86 file is

```
%B218A(80H, 00H)
```

This allows you to mount the iSBX 218A module in the SBX 1 socket of your CPU board.

3.4.4 %B224A Macro

The %B224A macro has the form

```
%B224A (instance, board_id, cylinders, heads, sectors, device_gran,
        slip$sectors, %(reserved))
```

where:

instance — A value indicating which iSBC 186/224A controller the driver should use if the system contains multiple iSBC 186/224A boards. During initialization the driver calculates the instance by scanning the MULTIBUS II slots in ascending order and sequentially assigning numbers to each iSBC 186/224A controller found. For example, 1 is assigned to the iSBC 186/224A board in the lowest-numbered slot, and 2 to the iSBC 186/224A in the next-lowest-numbered slot. This method of identifying the board provides slot independence.

board_id — A ten-byte string identifying the board. The board-id is found in registers 2-11 of the header record in the interconnect space. For the iSBC 186/224A controller board, the board_id is ASCII 186/224A followed by two ASCII NULL (0) characters and can be entered in the B224A.A86 file using the following form:

```
186/224AXX
```

where 'XX' are ASCII NULL (0) characters.

The following parameters are used for initializing Winchester disk drives but not floppy disk drives:

cylinders — A word specifying the number of cylinders on the disk.

heads — A byte specifying the number of fixed data heads on Winchester disk drives.

sectors — A byte specifying the number of sectors per track.

device_gran — A word specifying the number of bytes per sector for the device.

slip$sectors — A byte specifying the number of sectors per track to be used as alternate sectors when bad sectors are found during formatting. This feature is enabled only when the sector-slipping option is used. Currently sector-slipping is not supported; therefore, this value should be set to zero.

reserved — This parameter is reserved for future use. It consists of 10 one-byte values, separated by commas. The driver uses these bytes as the last ten bytes of the parameter buffer it uses to initialize the drive. For example, the iSBC 186/224A expects these ten bytes to be zero. This parameter may be specified as either

```
%(0,0,0,0,0,0,0,0,0,0)
```

or

```
%(10 dup(0))
```

The iSBC 186/224A device driver sends an initialize command to the iSBC 186/224A controller, which uses the preceding values to initialize the Winchester disk drive. Then the volume label is read. If the volume label has valid device characteristics, the drive is reinitialized with those characteristics.

Intel assumes the floppy disks are in standard format: track 0 formatted as 128 bytes/sector, 16 sectors/track. The disk characteristics are read from the volume label and the drive is reinitialized with those characteristics.

The default invocation of this macro in the B224A.A86 file is

```
%B224A ('186/224A??', 132H, 4, 9, 1024, 0,%(10 dup (0)))
```

Note, the characters '??' represent two ASCII NULL characters entered using AEDIT. To input an ASCII NULL character, invoke AEDIT, position the cursor on top of the second single quote mark, press the key 'H' for hex input, press the key 'I' for input,, press the key '0' for the value. After inserting one ASCII NULL character, enter a second one.

3.4.5 %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 invocation of this macro in the B251.A86 file is

```
%B251 (80H, 64)
```

3.4.6 %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 invocation of this macro in the B254.A86 file is

%B254 (0880H, 256, 8, 2048)

3.4.7 %B264 Macro

The %B264 macro has the form

%B264 (io_base, dev_gran, num_boards, board_size)

where:

io_base	I/O port address selected (jumpered) on the iSBC 264 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 264 board.

The default invocation of this macro in the B254.A86 file is

%B264 (0880H, 256, 4, 8192)

3.4.8 %BSCSI Macro

This macro allows you to specify the details of a SCSI host board, such as the iSBC 186/03A, iSBC 286/100 or iSBC 286/100A board, when an 8255A Programmable Peripheral Interface component is used to implement the host interface.

The %BSCSI macro has the form

```
%BSCSI (a_port, b_port, c_port, control_port, reserved, reserved,
        dma_controller, dma_channel, dma_base_address, dma_separation,
        scsi_info, info)
```

The END command at the end of this file is an ASM86 statement and it does not require a %.

where:

a_port — The WORD address of Port A of the 8255 Programmable Peripheral Interface (PPI) used by this SCSI driver.

b_port — The WORD address of Port B of the 8255 PPI used by this SCSI driver.

c_port — The WORD address of Port C of the 8255 PPI used by this SCSI driver.

control_port — The WORD address of the control word register of the 8255 PPI used by this SCSI driver.

reserved — Reserved for future use. It should be set to zero.

reserved — Reserved for future use. It should be set to zero.

dma_controller — The type of DMA controller used. Possible values are

Value	Controller Type
01	80186 DMA controller
02	82258 Advanced DMA controller

Other values are reserved for future use.

dma_channel — A BYTE that indicates which channel on the DMA controller will be used. Specify the number of the DMA channel as listed in the appropriate Intel data sheet.

dma_base_address — A WORD that indicates the base I/O port address of the controller's registers.

dma_separation — A BYTE that indicates the number of bytes separating consecutive ports on the controller.

scsi_info — This parameter is iSBC-board-specific; it does not depend on the SCSI driver's requirements. This parameter is a BYTE which has the following meaning:

Value	Meaning
0	Indicates that no additional information is needed to configure the Bootstrap Loader for the iSBC board you are using.
1	Indicates this configuration of the Bootstrap Loader is used on the iSBC 286/100A board.
2-255	Reserved for future use.

info — Varies depending on the value of scsi_info.

If scsi_info is 0, then no other information is needed and info is left blank.

If scsi_info is 1, then info is a single WORD that specifies the port address of the iSBC 286/100 or iSBC 286/100A port used for multiplexing DMA sources into the on-board 82258 DMA component.

The SCSI driver can be used to bootstrap load from any random-access device on the SCSI bus. The SCSI driver can also be used to bootstrap load from specific random-access devices on the SASI bus. When using the SASI bus, you must select a specific device, because the SASI devices require unique initialization information. Do this by specifying unique unit information for each device on the SASI bus (the %SASI_UNIT_INFO macro is used for this purpose).

The %BSCSI macro can be invoked only once in the BSCSI.A86 configuration file. The %SASI_UNIT_INFO macro (described in the next section) can be invoked multiple times to allow specification of the units on the SASI bus. Refer to the description of the %SASI_UNIT_INFO macro to see how to specify unique unit information for devices on the SASI bus.

In the BSCSI.A86 file, the default versions of the %BSCSI macro are

```
%BSCSI(0C8H, 0CAH, 0CCH, 0CEH, 0, 0, 1, 0, 0FFC0H, 2, 0)

;BSCSI(0C8H, 0CAH, 0CCH, 0CEH, 0, 0, 2, 0, 0200H, 2, 1, 0D1H)
```

The SCSI host board interface defined by the first instance (which is invoked) is the iSBC 186/03A board and uses the 80186 DMA controller.

The SCSI host board interface defined by the second instance (which is not invoked) is the iSBC 286/100 or iSBC 286/100A MULTIBUS II board and uses the on-board 82258 Advanced DMA controller. If you want to invoke this board, replace the ";" with a "%", and replace the "%" with a ";" to comment out the interface defined by the first instance (iSBC 186/03A board using the 80186 DMA controller).

An important feature to note about devices that use an SCSI controller is the configuration information is device-independent. That is, only the host board interface to the controller needs to be specified in the configuration file. The configuration values contain no information about the device(s) actually being used.

3.4.9 %SASI_UNIT_INFO Macro

The SCSI device driver provides an interface to mass storage devices through either SASI or SCSI controllers. If using devices controlled by a SASI controller, you must specify a sequence of initialization bytes for the controller. This information is not required by SCSI controllers. The initialization sequence identifies the type of device you have assigned to the particular unit of the SASI controller. The sequence will be different depending on the manufacturer and model of the hard disk or flexible diskette drive, and the manufacturer and model of the SASI controller board itself.

This macro enables you to define the initialization sequences required by your devices on the SASI bus. For each instance of the %DEVICE macro (in the BS1.A86 or BS1MB2.A86 file) that defines a device on the SASI bus, you must also include the %SASI_UNIT_INFO macro (in the %BSCSI.A86 file) to define that device's initialization sequence. The label specified for the unit info field of the %DEVICE macro must match the label field of the corresponding %SASI_UNIT_INFO field.

The information supplied by an occurrence of the %SASI_UNIT_INFO macro is not used by devices on the SCSI bus. Therefore in the BS1.A86 or BS1MB2.A86 file, %DEVICE macros for devices controlled by the SCSI bus should never specify a value for the unit info parameter. Note that there is only one pair of device initialization/device read procedures for the SCSI driver regardless of whether the controller is SCSI or SASI.

The %SASI_UNIT_INFO macro can be included only in the SCSI/SASI driver configuration file, BSCSI.A86. The macro has the form

```
%SASI_UNIT_INFO ( label, init_command, init_count, init_data )
```

where:

label	A valid ASM86 label name matching the one you specified in the unit info field of the %DEVICE macro for your device (in the file BS1.A86 or BS1MB2.A86).
init_command	A WORD that is the initialization command for your particular SASI controller.
init_count	A BYTE specifying the number of initialization BYTEs that your SASI controller requires.
init_data	The array of BYTEs of initialization data required by your SASI controller. The length of this array must be equal to the value in the init count parameter.

The default invocations of this macro in BSCSI.A86 are

```
;   iSBC 186/03A SCSI Host
%bscsi( 0C8H, 0CAH, 0CCH, 0CEH, 0, 0, 1, 0, 0FFC0H, 2, 0)
;
;   iSBC 286/100 SCSI Host
;bscsi( 0C8H, 0CAH, 0CCH, 0CEH, 0, 0, 2, 0, 0200H, 2, 1, 0D1H)
;
;   Xebec S1420 SASI controller and a Teac model F55B, 5 1/4-inch
;   flexible diskette drive.
%sasi_unit_info(sasi_x1420mf, 11h,10,0,28h,2,90h,3,0fh,50h,0fh,014h,0)
;
;   Xebec S1410 SASI controller and a Quantum model Q540, 5 1/4-inch
;   Winchester disk drive.
%sasi_unit_info(sasi_x1410b, 0ch, 8, 2, 0, 8, 2, 0, 0, 0, 0bh)
;
;   Xebec S1410 SASI controller and a Computer Memories, Inc.
;   model CMI-5619 5 1/4-inch Winchester disk drive.
%sasi_unit_info(sasi_x140a, 0ch, 8, 1, 32h, 6, 0, 0b4h, 0, 0, 0bh)
```

3.4.10 User-Supplied Drivers

If you want to bootstrap load your system from a device other than one for which Intel supplies a first stage device driver, you must write your own device initialization and device read device driver procedures that the first stage will call. Chapter 5 describes how to do this. In addition, perform the following actions to add the procedures to the Bootstrap Loader:

- Specify the names of the device initialization and device read procedures in a %DEVICE macro in the BS1.A86 or BS1MB2.A86 file.

- If there are configurable parameters associated with your device (such as base addresses or wakeup ports), you might want to create your own configuration macro and include it in a special configuration file, just like the Intel devices do. Chapter 5 describes how to set up a macro.
- Assemble your device initialization procedure, your device read procedure, and your configuration file (if you have one), and link the resulting object code to the rest of the Bootstrap Loader object files and libraries.

3.5 GENERATING THE FIRST STAGE

The submit file BS1.CSD performs the assembly, linkage, and location of the first stage of the Bootstrap Loader. Often it will need to be modified to generate the particular configuration of the Bootstrap Loader you specified in BS1.A86 or BS1MB2.A86. Figure 3-4 shows commands in the Intel-supplied BS1.CSD file.

```
attachfile $ as :f1:
;
;   The next four lines must be used to generate the Bootstrap Loader on
;   iRMX II.  The iRMX II Updates supply the MPL286 utility.
;
mpl286 :f1:%2.a86   $object(:f1:%2.mpl)
mpl286 :f1:bserr.a86 $object(:f1:bserr.mpl)
asm86 :f1:%2.mpl     macro(90) object(:f1:%2.obj)   print(:f1:%2.lst)
asm86 :f1:bserr.mpl  macro(50) object(:f1:bserr.obj) print(:f1:bserr.lst)
;
;   The next two lines must be used to generate the Bootstrap Loader on
;   iRMX 86.  No invocation of MPL286 is required.  Comment out the previous
;   four lines by inserting a ';' in front of the line.  Remove the ';' from
;   the front of the next two lines if generating on iRMX 86.
;
;asm86 :f1:%2.a86    macro(90) object(:f1:%2.obj)   print(:f1:%2.lst)
;asm86 :f1:bserr.a86  macro(50) object(:f1:bserr.obj) print(:f1:bserr.lst)
;
asm86 :f1:b208.a86   macro(50) object(:f1:b208.obj)  print(:f1:b208.lst)
asm86 :f1:bmsc.a86   macro(50) object(:f1:bmsc.obj)  print(:f1:bmsc.lst)
asm86 :f1:b218a.a86  macro(50) object(:f1:b218a.obj) print(:f1:b218a.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:b264.a86   macro(50) object(:f1:b264.obj)  print(:f1:b264.lst)
asm86 :f1:bscsi.a86  macro(50) object(:f1:bscsi.obj) print(:f1:bscsi.lst)
;
```

Figure 3-4. First Stage Configuration File BS1.CSD

```
;   Multibus II configuration
;
;asm86 :f1:b224a.a86  macro(50) object(:f1:b224a.obj) print(:f1:b224a.lst)
;
link86                                &
   :f1:%2.obj,                        &
   :f1:bserr.obj,                     &
&  :f1:bcico.obj,                     & ;for standalone serial channel support
   :f1:b208.obj,                      &
   :f1:bmsc.obj,                      &
   :f1:b218a.obj,                     &
   :f1:b251.obj,                      &
   :f1:b254.obj,                      &
   :f1:b264.obj,                      &
&  :f1:b224a.obj,                     &
   :f1:bscsi.obj,                     &
   :f1:bs1.lib                        &
to :f1:%2.lnk print(:f1:%2.mp1) &
&nopublics except(firststage,      &
&                 firststage_186,  &
&
&      Remove the '&' from the beginning of the previous line if the
&      iAPX_186_INIT macro is invoked in the configuration file.
&
&                 bootstrap_entry)
;
loc86 :f1:%2.lnk                                        &
   addresses(classes(code(0%0),stack(0%1)))             &
   order(classes(stack,data,boot,code,code_error))      &
&
   noinitcode                                           &
   start(firststage)                                    &
&
&      Change the previous line to 'start(firststage_186)' if the
&      iAPX_186_INIT macro is invoked in the configuration file.
&
   segsize(boot(1800H))                                 &
```

Figure 3-4. First Stage Configuration File BS1.CSD (continued)

```
   map print(:f1:%2.mp2)                                    &
   ;bootstrap
;
;       Remove the ';' from the line ';bootstrap' when generating a
;       a standalone Bootstrap Loader in PROM for a 80286-based CPU board.
;       Do not remove the ';' if the Bootstrap Loader is being generated
;       for an 80386-based CPU board.
;
;    Bootstrap Loader first stage generation complete.
;
```

Figure 3-4. First Stage Configuration File BS1.CSD (continued)

3.5.1 Modifying the BS1.CSD Submit File

To generate your own version of the Bootstrap Loader first stage, there are several changes you might need to make.

First, if you have excluded any device drivers from the Bootstrap Loader (by excluding %DEVICE macros from the BS1.A86 or BS1MB2.A86 file), you won't want to link the code for those drivers into the the first stage. To prevent the linking of a device driver, edit the LINK86 command in the BS1.CSD file and place an ampersand (&) in front of any file name that corresponds to a driver you want to exclude. Figure 3-5 is an example that shows a portion of the BS1.CSD file after excluding the iSBC 208, iSBX 218A, iSBX 251, iSBC 254 and SCSI device drivers.

```
link86                                &
   :f1:%2.obj,                        &
   :f1:bserr.obj,                     &
&  :f1:bcico.obj,                      & ;for standalone serial channel support
&  :f1:b208.obj,                       &
   :f1:bmsc.obj,                      &
&  :f1:b218a.obj,                      &
&  :f1:b251.obj,                       &
&  :f1:b254.obj,                       &
   :f1:b264.obj,                      &
   :f1:b224a.obj                      &
&  :f1:bscsi.obj,                      &
   :f1:bs1.lib                        &
to :f1:%2.lnk print(:f1:%2.mp1) &
&nopublics except(firststage,      &
&                 firststage_186,  &
&
&      Remove the '&' from the beginning of the previous line if the
&      iAPX_186_INIT macro is invoked in the configuration file.
&
```

Figure 3-5. Excluding the iSBC 251 and iSBC 254 Drivers

NOTE

If you exclude a device driver, do NOT include any %DEVICE macros for it in the BS1.A86 or BS1MB2.A86 configuration file or errors from LINK86 will occur.

Also, if you are not using an iRMX I or iRMX II system to configure the Bootstrap Loader, you must comment out the command attaching the directory where the Bootstrap Loader files reside as the logical name :F1:. Change the line:

```
ATTACHFILE $ AS :F1:
```

to

```
;ATTACHFILE $ AS :F1:
```

3.5.2 Invoking the BS1.CSD Submit File

After you have modified the BS1.CSD file to correspond to your configuration, invoke the submit file to assemble the Bootstrap Loader files, link them together, and assign absolute addresses. The format for invoking the submit file is as follows:

```
ATTACHFILE /RMX286/BOOT
SUBMIT BS1(first_stage_address, second_stage_address, first_stage_file)
```

where:

first_stage_address	The starting address of the first stage of the Bootstrap Loader. This can be a RAM address if you intend to run the Bootstrap Loader from RAM, or it can be a PROM devices address if you intend to place the Bootstrap Loader into PROM devices. The address you specify should be a full 20-bit address. Do not use the base:offset form to indicate the address. The iSDM Release 3.2 monitor allocates the address range from 0FE400H to 0FFF7FH to the Bootstrap Loader. If your configuration of the Bootstrap Loader will not fit in this space, locate it at a lower address than FF8000H.
second_stage_address	The address in RAM where the second stage of the Bootstrap Loader will be loaded. The data area for the first and second stages is also located here. The size of this second stage area consists of less than 8K contiguous bytes. The default address for the second stage is 0B8000H. This address has been chosen to be compatible with the default address of the third stage which is 0BC000H.
first_stage_file	The first-stage configuration file to use. If your system is a MULTIBUS I system, set this parameter equal to the string 'bs1'. Setting this parameter to 'bs1' causes the located Bootstrap Loader file to be named 'bs1'. If your system is a MULTIBUS II system, set this parameter equal to the string 'bs1mb2'. Setting this parameter to 'bs1mb2' causes the located Bootstrap Loader file to be named 'bs1mb2'.

To invoke the BS1.CSD SUBMIT file with the default addresses for combining with the iSDM monitor, type one of the two sets of commands below:

-
-

-
-

3.6 MEMORY LOCATIONS OF THE FIRST AND SECOND STAGES

When you invoke the BS1.CSD file, you assign memory locations to the first and second stages. It is important that the addresses you assign do not cause the stages to overlap, either with themselves or with the files they load. Chapter 4 discusses the memory locations of all three stages of the Bootstrap Loader and the steps to take to ensure that they don't overlap. Also inspect the map file, BS1.MP2, to ensure the segments are properly laid out. If too many device drivers have been configured into the Bootstrap Loader, some segments will be located in low memory starting at 200H. This is unacceptable and you must remove some more device drivers from your configuration.

CHAPTER 4
CONFIGURING THE THIRD STAGE

4.1 INTRODUCTION

The third stage of the Bootstrap Loader is used only for loading iRMX II systems. It provides the capability of loading modules that use the 80286 object module format (such as those produced using BND286 and BLD286) and those that require the processor's protected virtual address mode. This chapter describes how to configure the third stage.

There are two different types of third stages that can be used to load iRMX II files: the generic third stage and the device-specific third stage. Both load OMF-286 modules, but the generic third stage leaves the processor in real address mode while it loads. This permits it to use the first-stage device drivers to access the storage devices. The device-specific third stage switches the processor into protected mode before calling the device driver. Although this permits the device driver to load into the entire 16 megabyte address space, special device drivers that work in protected mode must be included in the third stage.

Configuration of the third stage differs slightly depending on whether you configure the generic or device-specific third stage. However, the differences are small enough that both will be described together throughout most of this chapter. The next two sections provide overviews of configuring each type of third stage. The rest of the chapter provides the details of third-stage configuration, noting any options that apply specifically to one type of third stage.

4.2 OVERVIEW OF THIRD STAGE CONFIGURATION

Configuring the third stage (either the generic or device-specific third stage) is very similar to configuring the first stage. It involves the following operations:

1. Editing an assembly language source file to indicate which CPU board to run on and what to do if errors occur during bootstrap loading. If you are using the device-specific third stage, you must also indicate which devices the third stage supports.
2. Invoking a SUBMIT file to assemble one or more assembly language source files, link them with code for the third stage, and assign absolute addresses to the code. This executable module remains in a file to be loaded by the second stage.

Like the first stage, the device-specific third stage requires its own device drivers. Therefore, you might expect to modify, assemble, and link configuration files for each of the devices, just as you do for the first stage. Actually, the SUBMIT file does assemble and link the device configuration files, but you don't need to do any additional work on these files. Because device-specific information (such as the I/O port address, the number of cylinders, etc.) is the same regardless of which stage accesses the device, the SUBMIT file uses the same device configuration files used for first-stage configuration.

The generic third stage uses the first-stage device drivers to communicate with mass storage devices. Therefore there is no need to supply configuration information about devices to the generic third stage.

Default versions of the assembly language source files and the SUBMIT file are placed in the /RMX286/BOOT directory during installation. These files include the following:

BS3.A86 BS3MB2.A86 BG3.A86	These assembly language source files contain macros that specify the devices supported by the third stage (for device-specific third stage only), identify the CPU board, and indicate what to do if errors occur during bootstrap loading. The BS3.A86 file applies to the device-specific third stage for MULTIBUS I systems, the BS3MB2.A86 file applies to the device-specific third stage for MULTIBUS II systems, and the BG3.A86 file applies to the generic third stage on either MULTIBUS I or MULTIBUS II systems.
BMSC.A86 B264.A86	These assembly language source files apply just to the device-specific third stage. They contain configuration information about the devices in your system. These are the same files that were used during the configuration of the first stage. You do not need to modify them for the device-specific third stage.
BS3.CSD BG3.CSD	These SUBMIT files contain the commands needed to assemble the source files, link the resulting modules (and any other you supply) with the code for the third stage, and locate the resulting object module. The BS3.CSD file applies to the device-specific third stage, while the BG3.CSD file applies to the generic third stage.

As shipped on the release diskettes, these files are set up to generate the default versions of the Bootstrap Loader's device-specific and generic third stages.

4.3 BS3.A86, BS3MB2.A86, AND BG3.A86 CONFIGURATION FILES

Figures 4-1, 4-2, and 4-3 list the assembly language configuration files for the device-specific third stage files BS3.A86 and BS3MB2.A86 and the generic third stage file BG3.A86. Each of these files consists of an INCLUDE statement and several macros. The definitions of the macros that can appear in these files are contained in the INCLUDE file (BS3CNF.INC). These macros are similar to the macros that can appear in the first stage configuration file.

To configure your own version of the generic or device-specific third stage, you should edit the BS3.A86, BS3MB2.A86, or BG3.A86 file to include or exclude macros. For each macro, a percent sign (%) preceding the name includes (invokes) the macro, and a semicolon (;) preceding the name excludes the macro, treating it as a comment.

NOTE

When you exclude a macro, you must replace the percent sign with a semicolon. Don't just add a semicolon in front of the percent sign.

The following sections describe the macros that can appear in the BS3.A86, BS3MB2.A86, and BG3.A86 files. Unless otherwise specified, the macros can appear in either of the three files (the %DEVICE macro is the only one that applies just to the device-specific third stage).

```
name     bs3

$include (:f1:bs3cnf.inc)
$include(:f1:bmps.inc)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                    Multibus I devices                         ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%device (0,w0,deviceinitmscgen,devicereadmscgen,data_msc)
%device (1,w1,deviceinitmscgen,devicereadmscgen,data_msc)
%device (8,wf0,deviceinitmscgen,devicereadmscgen,data_msc)
%device (9,wf1,deviceinitmscgen,devicereadmscgen,data_msc)
%device (0,s0,deviceinitscsi, devicereadscsi,data_scsi)
%device (0,sx1410a0,deviceinitscsi,devicereadscsi,data_scsi,sasi_x1410a)
%device (0,sx1410b0,deviceinitscsi,devicereadscsi,data_scsi,sasi_x1410b)
%device (2,smf0,deviceinitscsi,devicereadscsi,data_scsi,sasi_x1420mf)
%device (0,pmf0,deviceinit218Agen,deviceread218Agen,data_218)
%device (0,ba0,deviceinit264,deviceread264,data_264)
;
;int1
%int3
;halt
;
%cpu_board (286/12)
;
%end
```

Figure 4-1. Intel-Supplied BS3.A86 File

```
    name    bs3

$include (:f1:bs3cnf.inc)
$include(:f1:bmps.inc)
;
;  MPC and ADMA configuration for iSBC 286/100 with iEXM 100 MPC module
;bmps(00H, 4, 08BH, 200H, 3, 2, 0A0H, 16)
;
;  MPC and ADMA configuration for iSBC 286/100A
;bmps(00H, 4, 08BH, 200H, 2, 3, 0E0H, 16)
;
;  MPC and ADMA configuration for iSBC 386/100
%bmps(00H, 4, 089H, 200H, 2, 3, 000H, 16)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                    Multibus II devices                                    ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
%device (0,s0,deviceinitscsi, devicereadscsi,data_scsi)
%device (0,sx1410a0,deviceinitscsi,devicereadscsi,data_scsi,sasi_x1410a)
%device (0,sx1410b0,deviceinitscsi,devicereadscsi,data_scsi,sasi_x1410b)
%device (2,smf0,deviceinitscsi,devicereadscsi,data_scsi,sasi_x1420mf)
%device(0, w0, device_init_224a, device_read_224a, data_bs_drivers)
%device(1, w1, device_init_224a, device_read_224a, data_bs_drivers)
%device(2, wf0, device_init_224a, device_read_224a, data_bs_drivers)
%device(3, wf1, device_init_224a, device_read_224a, data_bs_drivers)
;
;int1
%int3
;halt
;
%cpu_board (386/100)
;
%end
```

Figure 4-2. Intel-Supplied BS3MB2.A86 File

```
name    bg3

$include (:f1:bs3cnf.inc)
;
;int1
%int3
;halt

%cpu board (286/12)

%installation(n)

%end
```

Figure 4-3. Intel-Supplied BG3.A86 File

4.3.1 %BMPS Macro (MULTIBUS® II Only)

The %BMPS macro configures the message passing system used during bootstrap loading. This macro identifies the base address of the Message Passing Coprocessor (MPC), address distance between MPC ports, and information that defines how direct memory access (DMA) transfers occur.

The syntax of the %BMPS macro is

%BMPS (mpc$base$addr, port$sep, duty$cycle, dma$base$addr, dmain, dmaout, dma$trans, data$width)

where:

mpc$base$addr	The base I/O port address of the MPC. Refer to the appropriate single board computer user's guide for this address.
port$sep	The number of addresses separating individual MPC ports. For example, if the mpc$base$addr is 0000H and the next three I/O port addresses are 0004H, 0008H, and 000CH, respectively, the port$sep is 4H. Refer to the appropriate single board computer user's guide for the I/O port address map.

duty$cycle	The MPC duty cycle for the local bus. (The rate at which data packets are generated.) For information on how to calculate a duty cycle suitable for the local bus, refer to the MPC User's Manual. For duty cycles suitable for Intel single board computers, refer to the appropriate single board computer user's guide.
dma$base$addr	The base I/O port address for the Advanced Direct Memory Access (ADMA) controller. Refer to the appropriate single board computer user's guide for this address.
dma$in	The channel used to receive (input) DMA message passing transfers. Refer to the appropriate single board computer user's guide for this channel number.
dma$out	The channel used to send (output) DMA message passing transfers. Refer to the appropriate single board computer user's guide for this channel number.
dma$trans	The I/O port address used for DMA data transfers. Refer to the appropriate single board computer user's guide for this address.
data$width	The data width in bits of the local bus. This value must be either 16 or 32 (decimal). If the width is set to 32 bits on a 386/116- or 386/120-based board, flyby (one cycle) mode is enabled.

The %BMPS macro can generate errors if the local bus width is not 16 or 32 bits wide.

4.3.2 %DEVICE Macro (BS3.A86 and BS3MB2.A86 Only)

The %DEVICE macro applies only to the device-specific third stage (BS3.A86 and BS3MB2.A86 files). It associates a device with a particular third stage device driver. The syntax of the %DEVICE macro is as follows:

%DEVICE (unit, name, device$init, device$read, device$data,unit_info)

where:

unit	The unit number of this device. Unit numbering should be the same as that used in the BS1.A86 or BS1MB2.A86 file described in Chapter 3.
name	The name of the device. You should always specify the same name that you used for the device in the BS1.A86 or BS1MB2.A86 file.

device$init — Public name of the third stage device driver's initialization procedure. Table 4-1 lists the names used for Intel-supplied device drivers. If you supply your own driver (written as described in Chapter 6), enter the name of its initialization procedure.

device$read — Public name of the third stage device driver's read procedure. Table 4-1 lists the names used for Intel-supplied device drivers. If you supply your own driver (written as described in Chapter 6), enter the name of its read procedure.

device$data — Public name of a label that marks the first byte of the data segment used by the third stage device driver. Table 4-1 lists the names used for Intel-supplied device drivers. If you supply your own driver (written as described in Chapter 6), you must create such a label and enter its name here.

unit_info — An ASM86 label that marks the location of an array of BYTEs containing specific device-unit information required by the mass storage device defined by this invocation of the %DEVICE macro.

Table 4-1 lists the names of the device initialization procedures, device read procedures, and data segments for Intel-supplied third stage device drivers.

Table 4-1. Names for Intel-Supplied Third Stage Drivers

Device Driver	Device Initialize Procedure	Device Read Procedure	Data Segment
MSC Driver	deviceinitmscgen	devicereadmscgen	data_msc
iSBC 264 Driver	deviceinit264	deviceread264	data_264
iSBC 186/224A Driver	device_init_224A	device_read_224A	data_bs_drivers
SCSI Driver	deviceinitscsi	devicereadscsi	data_scsi

4.3.3 %SASI_UNIT_INFO Macro (BSCSI.A86 File)

The SCSI device driver provides an interface to mass storage devices through either SASI or SCSI controllers. If using devices controlled by a SASI controller, you must specify a sequence of initialization bytes for the controller. This information is not required by SCSI controllers. The initialization sequence identifies the type of device you have assigned to the particular unit of the SASI controller. The sequence will be different depending on the manufacturer and model of the hard disk or flexible diskette drive, and the manufacturer and model of the SASI controller board itself.

This macro enables you to define the initialization sequences required by your devices on the SASI bus. For each instance of the %DEVICE macro (in the BS1.A86 or BS1MB2.A86 file) that defines a device on the SASI bus, you must also include the %SASI_UNIT_INFO macro (in the %BSCSI.A86 file) to define that device's initialization sequence. The label specified for the unit info field of the %DEVICE macro must match the label field of the corresponding %SASI_UNIT_INFO field.

The information supplied by an occurrence of the %SASI_UNIT_INFO macro is not used by devices on the SCSI bus. Therefore in the BS1.A86 or BS1MB2.A86 file, %DEVICE macros for devices controlled by the SCSI bus should never specify a value for the unit info parameter. Note that there is only one pair of device initialization/device read procedures for the SCSI driver regardless of whether the controller is SCSI or SASI.

The %SASI_UNIT_INFO macro can be included only in the SCSI/SASI driver configuration file, BSCSI.A86. The macro has the form

```
%SASI_UNIT_INFO( label, init_command, init_count, init_data )
```

where:

label	A valid ASM86 label name matching the one you specified in the unit info field of the %DEVICE macro for your device (in the file BS1.A86 or BS1MB2.A86).
init_command	A WORD that is the initialization command for your particular SASI controller.
init_count	A BYTE specifying the number of initialization BYTEs that your SASI controller requires.
init_data	The array of BYTEs of initialization data required by your SASI controller. The length of this array must be equal to the value in the init count parameter.

The default invocations of this macro in BSCSI.A86 are

```
;   iSBC 186/03A SCSI Host
%bscsi( 0C8H, 0CAH, 0CCH, 0CEH, 0, 0, 1, 0, 0FFC0H, 2, 0)
;
;   iSBC 286/100 SCSI Host
;bscsi( 0C8H, 0CAH, 0CCH, 0CEH, 0, 0, 2, 0, 0200H, 2, 1, 0D1H)
;
;   Xebec S1420 SASI controller and a Teac model F55B, 5 1/4-inch
;   flexible diskette drive.
%sasi_unit_info(sasi_x1420mf, 11h,10,0,28h,2,90h,3,0fh,50h,0fh,014h,0)
;
;   Xebec S1410 SASI controller and a Quantum model Q540, 5 1/4-inch
;   Winchester disk drive.
%sasi_unit_info(sasi_x1410b, 0ch, 8, 2, 0, 8, 2, 0, 0, 0, 0bh)
;
;   Xebec S1410 SASI controller and a Computer Memories, Inc.
;   model CMI-5619 5 1/4-inch Winchester disk drive.
%sasi_unit_info(sasi_x140a, 0ch, 8, 1, 32h, 6, 0, 0b4h, 0, 0, 0bh)
```

4.3.4 %INT1 Macro

The %INT1 macro causes the third stage to execute an INT 1 (software interrupt) instruction whenever a bootstrap loading error occurs. This enables you to pass control to a user-written program if loading fails. However, to pass control to another program, you must place the address of that program in position 1 of the interrupt vector table. This macro is supported by only the D-MON386 monitor. The iSDM monitor does not support this macro.

The syntax of the %INT1 macro is

%INT1

There are no parameters associated with this macro.

Exactly one of the %INT1, %INT3, and %HALT macros must be included, or an error will occur when BS3.A86, BS3MB2.A86, or BG3.A86 are assembled.

4.3.5 %INT3 Macro

The %INT3 macro causes the third stage to execute an INT 3 (software interrupt) instruction whenever a bootstrap loading error occurs. If you are using the iSDM monitor, the INT 3 instruction passes control to the monitor. Otherwise, the INT 3 instruction has no effect unless you have placed the address of your custom interrupt handler in position 3 of the interrupt vector table.

The syntax of the %INT3 macro is

```
%INT3
```

There are no parameters associated with this macro.

Exactly one of the %INT1, %INT3, and %HALT macros must be included, or an error will occur when BS3.A86, BS3MB2.A86, or BG3.A86 are assembled.

4.3.6 %HALT Macro

The %HALT macro causes the third stage to execute a halt instruction whenever a bootstrap loading error occurs. The syntax of the %HALT macro is as follows:

```
%HALT
```

There are no parameters associated with this macro.

Exactly one of the %INT1, %INT3, and %HALT macros must be included, or an error will occur when BS3.A86, BS3MB2.A86, or BG3.A86 are assembled.

4.3.7 %CPU_BOARD Macro

The %CPU_BOARD macro specifies the type of processor board in your system. The third stage needs this information so that it can properly initialize the board when switching into protected virtual address mode. The syntax of the %CPU_BOARD macro is as follows:

```
%CPU_BOARD (type)
```

where:

type — The type of processor board in your system. The following are the valid values:

Value	Processor Board
286/12	iSBC 286/10 board
286/12	iSBC 286/10A board
286/12	iSBC 286/12 board
286/100A	iSBC 286/100A board
386/20	iSBC 386/2X board or iSBC 386/3X Board
386/100	iSBC 386/116 board or iSBC 386/120 Board

4.3.8 %INSTALLATION Macro (BG3.A86 Only)

The %INSTALLATION macro specifies whether the generic third stage will enter the monitor after loading the application system or not. The syntax of the %INSTALLATION macro is:

%INSTALLATION (monitor_entry)

where:

monitor_entry — The type of action the Bootstrap Loader is to take upon loading the application system. If monitor entry is 'n' the system is loaded and then executed with no monitor entry inbetween. If it is 'y', the monitor is entered after the system is loaded. You must type in the monitor GO command to continue.

When the monitor is entered, as a result of specifying 'y' for the monitor_entry parameter, the Bootstrap Loader prints the following message to the terminal:

```
Insert the Start-up System Commands Diskette and type "G<RETURN>"
```

NOTE

If your system has the D-MON386 monitor rather than the iSDM monitor, type "GO<RETURN>".

This macro is used to generate the generic third stage used to boot the Operating System from diskettes. The %INSTALLATION macro allows one diskette, which contains only the Operating System boot file and the third stage to be used to load the system from diskette into memory. In entering the monitor, it allows a second diskette, which contains the necessary system commands, to be used as the system device when the system is initialized.

4.3.9 %END Macro

The %END macro is required at the end of the BS3.A86, BS3MB2.A86, and BG3.A86 files. The syntax of this macro is as follows:

```
%END
```

There are no parameters associated with the %END macro.

4.3.10 User-Supplied Drivers

If you want to use the device-specific third stage to load your system from a device other than one for which Intel supplies a third-stage driver, you must write your own device driver procedures that the third stage will call. Chapter 6 describes how to do this. In addition, perform the following actions to add the procedures to the Bootstrap Loader:

- Specify the names of the device initialization procedure, the device read procedure, and the driver's data segment in a %DEVICE macro in the BS3.A86 file.
- If there are configurable parameters associated with your device (such as base addresses or wakeup ports), you might want to create your own configuration macro and include it in a special configuration file, just like the Intel devices do. Chapter 5 describes how to set up a macro. You will probably use the same configuration file for both the first- and third-stage drivers.
- Assemble your device initialization procedure, your device read procedure, and your configuration file (if you have one), and link the resulting object code to the rest of the Bootstrap Loader object files and libraries.

4.4 GENERATING THE THIRD STAGE

Two SUBMIT files (BS3.CSD and BG3.CSD) are used to generate the two types of third stages. BS3.CSD performs the assembly, linkage, and location of the device-specific third stage. BG3.CSD performs the same operations for the generic third stage. Figures 4-4 and 4-5 show the Intel-supplied BS3.CSD and BG3.CSD files.

```
attachfile $ as :f1:
;
mpl286 :f1:%0.a86 $object(%0.mpl)
asm86 :f1:%0.mpl
asm86 :f1:bmsc.a86
asm86 :f1:b218a.a86
asm86 :f1:b264.a86
asm86 :f1:bscsi.a86
asm86 :f1:b224a.a86
;
link86              &
   :f1:%0.obj,      &
   :f1:bs3.lib,     &
   :f1:bmsc.obj,    &
   :f1:b218a.obj,   &
   :f1:b264.obj     &
&  :f1:bscsi.obj,   &
&  :f1:b224a.obj    &
to :f1:%0.lnk print(:f1:%0.mpl) notype nolines nosymbols
;
loc86 :f1:%0.lnk                        &
    addresses(classes(code(%1)))        &
    order(classes(code,data))           &
    noinitcode purge                    &
    start(bs3)                          &
    map print(:f1:%0.mp2)
;
;   The Third Stage, located at address %1, is in the file %0
;
```

Figure 4-4. Device-Specific Third Stage SUBMIT File (BS3.CSD)

```
attachfile $ as :f1:
;
asm86   :F1:bg3.a86     nolist
;
link86                  &
    :f1:bg3.obj,        &
    :f1:bg3.lib         &
to  :f1:bg3.lnk notype nolines nosymbols
;
loc86   :f1:bg3.lnk                     &
    addresses(classes(code(%1)))        &
    order(classes(code,data))           &
    noinitcode                          &
    start(bs3) purge                    &
    to  :F1:%0.%1 map print(:f1:%0.mp2)
delete  :F1:bg3.tmp
;
;The Generic Third Stage is located at address %2 and is
;in the file %0.%1.
;
```

Figure 4-5. Generic Third Stage SUBMIT File (BG3.CSD)

4.4.1 Modifying the Submit Files

Before generating your own version of the third stage, you should modify the appropriate submit file to match your intended configuration.

If you are using the device-specific third stage and you have excluded any device drivers from it (by excluding %DEVICE macros from the BS3.A86 or BS3MB2.A86 file), you won't want to link the code for those drivers into the the third stage. To prevent the linking of a device driver, edit the LINK86 command in the BS3.CSD file and place an ampersand (&) in front of any file name that corresponds to a driver you want to exclude.

If you are not using an iRMX I or iRMX II system to configure the third stage, you must comment out the line where the directory containing the Bootstrap Loader files is attached as :f1: before invoking the other commands in the BS3.CSD or BG3.CSD file. Change the line:

```
ATTACHFILE $ AS :F1:
```

to

```
;ATTACHFILE $ AS :F1:
```

4.4.2 Invoking the Submit File

After you have modified either the BS3.CSD or BG3.CSD file to correspond to your configuration, invoke the appropriate SUBMIT file to assemble the third stage files, link them together, and assign absolute addresses. The format for invoking either SUBMIT file is as follows:

Device-specific third stage

SUBMIT BS3 (filename, third_stage_addr)

Generic third stage

SUBMIT BG3 (filename, extension, third_stage_addr)

where:

filename	The name of the file in which to store the generated third stage. Also, the name of the third-stage configuration file you are using (BS3.A86 for MULTIBUS I systems and BS3MB2.A86 for MULTIBUS II systems). The generic third stage appends the next parameter (extension) to the filename.
extension	The extension the generic third stage is to have. This does not apply to the device specific third stage. Normal generic third stages usually have the extension 'GEN'. Generic third stages used for Operating System installation should use the extension 'INS'.
third_stage_addr	The address in RAM where the third stage will be loaded. The address you specify should be a full 20-bit address. Do not use the base:offset form to indicate the address. If you have no special requirements for loading the third stage, specify a value of 0BC000H for this parameter.

4.5 MEMORY LOCATIONS OF THE THREE STAGES

When you configure the first and third stages of the Bootstrap Loader, you can assign the addresses at which the three stages will be located. Before setting these addresses, you must understand how default memory is assigned in the Bootstrap Loader.

Table 4-2 lists the default memory locations used by the Bootstrap Loader. It also names the SUBMIT files you can invoke to change the memory assignments.

Table 4-2. Memory Locations Used by the Bootstrap Loader

Description	Default	Maximum Size*	Configuration File
1st Stage Code	Application Dependent * 0FE400H for iSDM R3.0	14K Bytes	BS1.CSD
2nd Stage Code, 1st/2nd Data and Stack	0B8000H	8K Bytes	BS1.CSD
3rd Stage (specific) Code, Data and Stack	0BC000H	16K Bytes	BS3.CSD
3rd Stage (generic) Code	0BC000H	8K Bytes	BG3.CSD
Third Stage (generic) Data and Stack	0B8000H	—	BS1.CSD

* Maximum size is a function of the size of the device drivers included in the Bootstrap Loader.

The Bootstrap Loader Release Diskette contains a standalone version of the Bootstrap Loader, in the file named BS1, which selects all the supported Intel device drivers. The map file, BS1.MP2, is supplied to show the layout of the segments in BS1. The first stage is located at 0C0000H and the second stage is located at 0B8000H. All default third stages are located at 0BC000H.

CHAPTER 5
WRITING A CUSTOM FIRST-STAGE DRIVER

5.1 INTRODUCTION

You can configure the Bootstrap Loader to run with many kinds of devices. If you plan to use one of the devices for which Intel supplies a device driver, you can 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 first-stage device driver. (If you want to load iRMX II applications past the first megabyte of address space, you must also write a custom third-stage driver. Chapter 6 describes how to write third-stage drivers.) This chapter provides you with guidelines for writing a custom first-stage driver.

You must include two procedures in every first-stage device driver: a device initialize procedure and a device read procedure. The device initialize procedure must initialize the bootstrap device. The device read procedure must load 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, provided no other first-stage driver procedure uses the chosen names. To check the names of the Intel-supplied first-stage procedures, use LIB86 to list the modules in the object library /RMX286/BOOT/BS1.LIB or /RMX86/BOOT/BS1.LIB.

You must write both procedures in an 8086 language (either PL/M-86 or ASM86) and conform to the LARGE model of segmentation of the PL/M-86 programming language. This means that you must declare the two procedures as FAR (not NEAR) and all pointers must be 32 bits long. You must adhere to the interfacing and referencing conventions of the PL/M-86 LARGE model even if you write the procedures in assembly language.

If your driver code is going to operate in the MULTIBUS II environment, two additional driver code constraints exist. First, you must follow the MULTIBUS II transport protocol for communication between the driver and the device controller you bootstrap load from. You can accomplish this by using Bootstrap Loader Communication System utility calls within your driver code. Second, you must organize your driver code so that it belongs to the BSL-Drivers COMPACT sub-system. This last requirement is necessary because the Bootstrap Loader Communication System utilities are all NEAR calls.

The next two sections describe the interface these two procedures must present to the first-stage Bootstrap Loader code. Subsequent sections describe how to supply configuration information to the driver, how to use Bootstrap Loader Communication System utilities in your driver code, and how to generate first-stage Bootstrap Loader code that includes the new driver.

5.2 DEVICE INITIALIZE PROCEDURE

The device initialize procedure must present the following PL/M-86 interface to the Bootstrap Loader:

```
device$init: PROCEDURE (unit) WORD PUBLIC;
DECLARE unit        WORD;
    .
    .   (Typical code)
    .
END device$init;
```

where:

device$init	The name of the device initialize procedure. You can choose any name you wish for this procedure, as long as it does not conflict with the names of any other first-stage procedure.
unit	The device 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.

To be compatible with the Bootstrap Loader, the device initialize procedure must perform the following steps:

1. Test to see if the device is present. If not, return the value zero.
2. Initialize the device for reading. This operation is device-dependent. For guidance in initializing the device, refer to the hardware reference manual for the device.
3. Test to see if device initialization is successful. If not, return the value zero.
4. Obtain the device granularity. For some devices, only one granularity is possible, while for other devices several granularities are possible. The hardware reference manual for your device explains this device-dependent issue.
5. Return the device granularity.

NOTE

In addition to the above five steps, the procedure must follow MULTIBUS II transport protocol and belong to the BSL-Drivers COMPACT sub-system if the driver functions in a MULTIBUS II environment. Refer to Section 5.5 for more information on these two requirements.

5.3 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$ptr             POINTER;
        .
        .   (Typical code)
        .
END device$read;
```

where:

device$read	The name of the device read procedure. You can choose any name you wish for this procedure, as long as it does not conflict with the names of any other first-stage procedure.
unit	The device unit number, as defined during Bootstrap Loader configuration.
blk$num	A 32-bit number specifying the block number the Bootstrap Loader wants the procedure to read. The size of each block equals the device granularity, with the first block on the device being block number 0.
buf$ptr	A 32-bit POINTER to the buffer in which the device read procedure must copy the information it reads from the secondary storage device.

The device read procedure does not return a value to the caller.

To be compatible with the Bootstrap Loader, the device read procedure must perform the following steps:

1. Read the block specified by blk$num from the bootstrap device specified by unit into the memory location specified by buf$ptr.

2. Check for I/O errors. If none occur, 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, and call BSERROR. If no device code exists, use 00.

   Adding the following statements accomplish this in PL/M-86:

```
DECLARE BSERROR EXTERNAL;
DECLARE IO_ERROR LITERALLY '0B301H';
CALL BSERROR(IO_ERROR);
```

   If you call 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
```

NOTE

In addition to the above two steps, the procedure must follow MULTIBUS II transport protocol and belong to the BSL-Drivers COMPACT sub-system if the driver functions in a MULTIBUS II environment. Refer to Section 5.5 for more information on these two requirements.

5.4 SUPPLYING CONFIGURATION INFORMATION TO THE FIRST-STAGE DRIVER

Any custom device driver you write needs some configuration information about the device it supports, such as the address of the device wakeup port. (To determine what device-specific information your driver needs, consult the hardware reference manual for the device.) You can provide this information to the custom device driver one of two ways:

- Place the information directly into the driver (hard-coding)
- Create a configuration file similar to those provided with the Intel-supplied drivers.

5.4.1 Hard-Coding the Configuration Information

One way to supply configuration information to a custom device driver is to place it directly into the code. This method works, but if any of the configuration information changes, or if you want to support a similar device that has a slightly different configuration, you must change the driver and reassemble it. Fortunately, first-stage device drivers are usually small enough so that the amount of time required to reassemble them is negligible.

Figure 5-1 illustrates how to place the configuration information directly into the code. This figure lists the "Constants and Data" section that could be used to supply the MSC first-stage driver with device-specific configuration information.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;               Constants and Data
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;

data_bsmsc      segment     ;Static Data

wakeup_newdrivl dw  100H    ;MSC wakeup address

device_newdrivl db  0

drtab_newdrivl  dw  256     ;number of cylinders
    db              2       ;number of fixed heads
    db              0       ;number of removable heads
    db              9       ;number of sectors
    db              1024    ;device granularity
    db              5       ;number of alternate cylinders
```

Figure 5-1. Hard-Coded Configuration Information

5.4.2 Providing a Configuration File

The second way to supply configuration information is to declare all device-specific parameters as variables that are external to your device driver. A separate small module can declare these parameters as public variables in . You can incorporate this second module into the Bootstrap Loader by placing assembly and link commands in the first-stage SUBMIT file BS1.CSD. To use this approach, follow the steps below:

1. In the code for the device driver, declare the device-specific parameters as external variables. For example, the following code could be used instead of the hard-coding shown in Figure 5-1.

```
    name bpmsc
;Configuration information:
;
    extrn   wakeup_newdriv1 word    ;Wakeup port
                                    ;address
    extrn   device_newdriv1 byte    ;Device number
    extrn   drtab_newdriv1  byte    ;Device Table
```

2. Create an INCLUDE file containing a macro definition. The macro definition must declare the device-specific parameters as public variables (matching the external declarations from the previous step). This file should be named as "xxx.inc" where xxx is any name you choose. For example, you could place the following code into a file called NEWDRIV1.INC to define a macro for the device-specific parameters declared in Step 1.

```
%*DEFINE (bnewdriv1(wakeup,ncyl,nfsur,nrsur,nsec,secsize,nalt)) (
            name bnewdriv1
            public  wake_msc, device_msc, drtab_msc
code_newdriv1   segment byte    public  'CODE'
wakeup_newdriv1 dw      %wakeup

device_newdriv1 db      0

drtab_newdriv1  dw      %ncyl
                db      %nfsur
                db      %nrsur
                db      %nsec
                dw      %secsize
                db      %nalt

code_newdriv1   ends)

 %* DEFINE (end) (end)
```

3. Create another file that contains the macro invocation. You should name this file "xxx.a86", where xxx is any name you choose. The file must also contain an INCLUDE directive for the INCLUDE file created in the previous step. To be consistent with the Intel-supplied device drivers, the INCLUDE directive should use the logical name :F1: as a prefix to the name of the include file. For example, the file NWDRV1MAC.A86 could contain the following information to invoke the macro defined in Step 2.

```
$include(:f1:newdrivl.inc)

%bnewdrivl(100H, 256, 2, 0, 9, 1024, 5)
%end
```

If the device-specific configuration information ever changes, you can change the macro invocation in this file to reflect those changes. This is normally easier than changing the source code of the driver, especially for users who are not familiar with assembly language.

4. Store the files created in Steps 2 and 3 in the directory where the Bootstrap Loader configuration files reside (normally /RMX286/BOOT or /RMX86/BOOT). For example, the following Human Interface commands can be used to copy the files created in Steps 2 and 3.

```
  - copy newdrivl.inc, nwdrvlmac.a86 to &
** /rmx286/boot/newdrivl.inc, /rmx286/boot/nwdrvlmac.a86
newdrivl.inc copied to /rmx286/newdrivl.inc
nwdrvmac.a86 copied to /rmx286/nwdrvmac.a86
```

5. Edit the first-stage SUBMIT file (BS1.CSD) to cause it to assemble your configuration file and link it to the first stage. To the list of ASM86 invocations, add an ASM86 invocation for the file created in Step 3 (xxx.a86). To the list of modules to be linked (immediately below the LINK86 invocation), add the name of the object module created when your file (xxx.a86) is assembled. In both the ASM86 invocation and the LINK86 invocation, preface the filename with the logical name :F1: (such as :f1:xxx.a86). Unless you have reason to do otherwise, use the same ASM86 and LINK86 options shown for other files assembled and linked by BS1.CSD.

   Figure 5-2 shows modifications to BS1.CSD that add support for the driver configuration files just created. Arrows at the left of the figure show the lines that were added. Notice that only the configuration file is being assembled each time BS1.CSD is invoked, not the entire driver. BS1.CSD assumes the use of the configuration file BS1.A86 and that you have assembled your driver and added the resulting object module into the library BS1.LIB.

```
    ;
    ;asm86 :f1:bs1.a86  macro(90) object(:f1:bs1.obj)       print(:f1:bs1.lst)
            .
            .
            .
    asm86 :f1:b264.a86  macro(50) object(:f1:b264.obj)  print(:f1:b264.lst)
    asm86 :f1:bscsi.a86 macro(50) object(:f1:bscsi.obj) print(:f1:bscsi.lst)
--> asm86 :f1:nwdrv1mac.a86 macro(50) object(:f1:nwdrv1mac.obj nolist
    ;
    link86                          &
        :f1:bs1.obj,                &
        :f1:bserr.obj,              &
    &   :f1:bcico.obj,              &;standalone serial channel support
            .
            .
            .
        :f1:bscsi.obj,              &
        :f1:b264.obj,               &
-->     :f1:nwdrv1mac.obj,          &
        :f1:bs1.lib                 &
    to :f1:bs1.lnk print(:f1:bs1.mpl) &
            .
            .
            .
```

Figure 5-2. Modified BS1.CSD File

5.5 USING THE MULTIBUS® II TRANSPORT PROTOCOL

If the driver you are creating functions within a MULTIBUS I environment, you need not read this section. Skip to Section 5.6.

If the driver you are creating functions within a MULTIBUS II environment, you must write the driver code to use the MULTIBUS II message transport protocol. To help you accomplish this task, Intel provides a small, single-thread communication system that enables Bootstrap Loader drivers to communicate with device controllers within a MULTIBUS II environment. This communication system is called the Bootstrap Loader Communication System.

The following paragraphs provide an overview of the Bootstrap Loader Communication System, which uses concepts similar to the Nucleus Communication System. Should you desire a more complete description of these communication system concepts, refer to the *Extended iRMX II Nucleus User's Guide* in Volume 2 of the iRMX II documentation set.

The Bootstrap Loader Communication System can be thought of as a subset of the Nucleus Communication System. It fully conforms to the MULTIBUS II transport protocol suitable for a limited bootloading environment. Unlike the Nucleus Communication System, the Bootstrap Loader Communication system is designed to handle bootstrap loading only. Consequently, the system is synchronous in nature. In other words, procedures execute to completion one after the other; no multitasking or need to handle asynchronous events exists.

MULTIBUS II transport protocol functions supported by the Bootstrap Loader Communication System include control and data message types, a subset of the request/response transaction model, send and receive transaction models, message broadcasting, and access to device interconnect space.

To support these functions, Intel supplies a set of system utilities grouped together in a Bootstrap Loader Message Passing System Module. As a programmer, you have access to these utilities through system calls you place in your driver code. The remainder of this section explains the supported functions in the Bootstrap Loader Communications System and shows you how to use each of the utilities.

5.5.1 Message Passing Controller Initialization

Before any Bootstrap Loader Communication System calls can be made, you must initialize certain parts of the hardware in preparation for message passing. You accomplish this initialization through the BSMPSINIT utility. You must make this call from your driver's initialization procedure before making any other Bootstrap Loader Communication utility calls. The following utility description presents BSMPSINIT:

CALL BSMPSINIT

INPUT PARAMETERS

This utility has no input parameters.

OUTPUT PARAMETERS

This utility has no output parameters.

DESCRIPTION

The BSMPSINIT utility provides hardware initialization for the Message Passing Controller (MPC) and the Advanced Direct Memory Access (ADMA) devices. You must perform a call to this utility before attempting any other Bootstrap Loader Communication System utility calls.

CONDITION CODES

This utility has no condition codes.

5.5.2 Message Types

The Bootstrap Loader Communication System supports two types of messages: control messages and data messages.

Control messages consist of only a control portion. These messages occur between the sender and receiver requiring no explicit buffer resource allocation. The reason for no buffer allocation is because a control message has no data part. The maximum length of a control message is 20 bytes. Also, a one-to-one correspondence exists between control messages and MULTIBUS II unsolicited messages (all unsolicited messages are control messages).

Data messages consist of both a 16-byte control portion and a variable length data portion. These messages do require explicit buffer allocation between the sender and receiver. The reason buffer allocation is required is because this type of message contains a variable amount of data. The maximum length of the data portion is 64K-1 bytes.

5.5.3 Request/Response Transaction Model

The Bootstrap Loader Communication System supports a subset of the request/response transaction model that the Nucleus Communication System uses. This subset has the following characteristics:

- Because the Bootstrap Loader Communication System functions within a bootloading environment, request messages originate only from the host CPU board. The specific device controllers then match responses to requests on a one-to-one basis.
- No support exists for multiple outstanding requests.
- Fragmentation and transmission of response messages into specific application buffers can occur. Because this fragmentation is completely transparent to the user, the fragmented response is considered as a single response to a single request.
- The Bootstrap Loader Communication System receives messages in the order in which they are sent.

Communication between the CPU host board executing the driver and the bootable device controller uses the basic transmission model of send and receive. The driver sends a request to the device controller and then receives a response back. When the driver initiates the message, an internal transaction ID is generated that logically associates the request with the response. This ID remains valid until the device controller responds, thus completing the transaction.

For messages that require data as part of the response, the driver can initiate the allocation of an rsvp data message buffer in which to receive the response data. The data can then arrive either whole or fragmented. Regardless of fragmentation, the host CPU board views the response message as one message. If the request message requires no data as a response, the response must be a control message.

The utility the Bootstrap Loader Communication System uses to support the request/response transaction model is BS$SEND$RSVP. The following utility description presents BS$SEND$RSVP:

CALL BS$SEND$RSVP	(socket,control$ptr,data$adr,data$length, rsvp$control$p,rsvp$data$adr,rsvp$data$length, flags,exception$ptr)

INPUT PARAMETERS

socket	A DWORD of the form host$id:port$id identifying the remote destination.
control$ptr	A POINTER to a control message. If data$adr = NULL (0) or data$length = 0, then the control message is 20 bytes long. Otherwise, the control message is 16 bytes long.
data$adr	A DWORD containing the absolute address of a data message. If data$adr is NULL (0), then a control message is sent. Otherwise, data$adr points to a contiguous buffer.
data$length	A WORD defining the length of the data message. If data$length is equal to zero, the control message length is assumed to be 20 bytes.
rsvp$control$p	A POINTER to the received control message. If rsvp$data$adr = NULL (0) or rsvp$data$length = 0, then the control message is 20 bytes long. Otherwise, the control message is 16 bytes long.
rsvp$data$adr	A DWORD containing the absolute address of a data message buffer for the return response that is expected. If rsvp$data$adr is NULL (0), then a control message is expected as a reply. Otherwise, rsvp$data$adr points to a contiguous buffer in which the data message arrives.

rsvp$data$length	A WORD defining the length of the rsvp data buffer.
flags	WORD reserved for future use. Although this parameter is ignored, you must supply a "0" value as a placeholder.

OUTPUT PARAMETER

exception$ptr	A POINTER to a WORD to which the Operating System returns the exception code generated by this Bootstrap Loader Communication System call.

DESCRIPTION

The BS$SEND$RSVP utility sends a message from a port to a remote socket with an explicit request for a return response. This call is synchronous with respect to both the request and the response.

EXAMPLE

This example illustrates the fundamentals of the request/response transaction model between the host CPU board and bootable device controller board. This example is written in PL/M-86 code and is intended to be generic in nature.

```
/*************************************************************
 *     This example sends a 20-byte control message to the   *
 *     bootable device controller board located in slot 1    *
 *     at port 500 of the MULTIBUS II system.  This message  *
 *     solicits data from the device as part of the          *
 *     response.                                             *
 *                                                           *
 *     The control message sent is contained in the 20-byte  *
 *     data array p$command$msg (Peripheral Command          *
 *     Message).  The control message received is captured   *
 *     in the 20-byte data array p$status$msg (Peripheral    *
 *     Status Message).                                      *
 *                                                           *
 *     The solicited data is received from the device via    *
 *     an rsvp buffer.  Note that the address pointing to    *
 *     the rsvp buffer must be an absolute address before    *
 *     it is passed to BS$SEND$RSVP.  Thus, the need for     *
 *     calling a conversion routine.  In this example, a     *
 *     routine (not shown) called CONVERT_ADDRESS handles    *
 *     the address conversion.  It is up to the programmer   *
 *     to supply the conversion routine.                     *
 *                                                           *
 *     Setting data$length and data$adr to NULL (0)          *
 *     indicates that only a control message is being sent   *
 *     from the host CPU board to the controller board.      *
 *************************************************************/
```

```
SAMPLE_BS$SEND$RSVP: DO;

    DECLARE socket                DWORD;
    DECLARE socket$o structure
            (host$id              WORD,
            port$id               WORD) AT (@socket);
    DECLARE p$control$msg(20)     BYTE;
    DECLARE p$status$msg(20)      BYTE;
    DECLARE send$data(100)        BYTE;
    DECLARE rsvp$data(1024)       BYTE;
    DECLARE rsvp$data$adr         DWORD;
    DECLARE rsvp$data$length      DWORD;
    DECLARE flags                 WORD;
    DECLARE exception             WORD;
    DECLARE slot    LITERALLY     '1H';
    DECLARE port    LITERALLY     '1F4H';
    DECLARE null    LITERALLY     '0H';

    CODE: DO;
            socket$o.host$id = slot;
            socket$o.port$id = port;
            rsvp$data$length = 400H;
            flags            = null;

            rsvp$data$adr = CONVERT_ADDRESS (@rsvp$data(0));

                    .
                    .  (Typical code to define
                    .   the 20-byte p$control$msg block
                    .   with the control message.)
                    .

            CALL BS$SEND$RSVP
                (socket,@p$control$msg(0),null,
                null,@p$status$msg(0),
                rsvp$data$adr,rsvp$data$length,
                flags,@exception);

            IF exception <> 0
                THEN CALL BSERROR;
                    .
                    .  (Typical code to execute
                    .   for successful status.)
                    .
    END CODE;
END SAMPLE_BS$SEND$RSVP;
```

CONDITION CODES

E$OK	0000H	No exceptional conditions.
BSEBUFFER$SIZE	00E2H	The rsvp buffer posted is too small.
BSETRANSMISSION	00E1H	An error occurred while transmitting a MULTIBUS II message.

5.5.4 Send and Receive Transaction Models

In addition to the request/response transaction model, the Bootstrap Loader Communication System supports send and receive transaction models. Normally, communication between a driver and a device in a bootloading environment uses the request/response or send models. However, if your host CPU board can capitalize on a receive transaction model initiated from the driver, the utility is available.

You can make calls to the send and receive utilities, respectively when you need the driver to simply send a message with no request for a response, or when you need the driver to wait for spontaneous communication from a specific device controller.

The two utilities available to you that support the send and receive transaction models are BS$SEND and BS$RECEIVE. The following utility descriptions present BS$SEND and BS$RECEIVE:

CALL BS$SEND (socket,control$ptr,data$adr,data$length, flags,exception$ptr)

INPUT PARAMETERS

socket	A DWORD of the form host$id:port$id identifying the remote destination.
control$ptr	A POINTER to a control message. If data$adr = NULL (0) or data$length = 0, then the control message is 20 bytes long. Otherwise, the control message is 16 bytes long.
data$adr	A DWORD containing the absolute address of a data message. If data$adr is NULL (0), then a control message is sent. Otherwise, data$adr points to a contiguous buffer.

data$length	A WORD defining the length of the data message. If data$length is equal to zero, the control message length is assumed to be 20 bytes.
flags	WORD reserved for future use. Although this parameter is ignored, you must supply a "0" value as a placeholder.

OUTPUT PARAMETER

exception$ptr	A POINTER to a WORD in which the Bootstrap Loader returns the exception code generated by this Bootstrap Loader Communication System call.

DESCRIPTION

The BS$SEND utility sends either a control or a data message to a MULTIBUS II board identified by the parameter socket.

EXAMPLE

This example illustrates the fundamentals of message passing from the host CPU board to the bootable device controller board. This example is written in PL/M-86 code and is intended to be generic in nature.

```
/*************************************************************
 *     This example sends a data message to the bootable     *
 *     controller board located in slot 1 at port 500 of     *
 *     the MULTIBUS II system.                               *
 *                                                           *
 *     The control portion of the message sent is located    *
 *     in the 16-byte data array p$control$msg (Peripheral   *
 *     Command Message).  The data portion of the message    *
 *     sent is located in the 100-byte data array            *
 *     send$data.                                            *
 *                                                           *
 *     Note that the programmer is responsible for ensuring  *
 *     p$control$msg and the area containing the data        *
 *     portion of the message are initialized with correct   *
 *     data.                                                 *
 *************************************************************/
```

```
SAMPLE_BS$SEND: DO;

    DECLARE socket              DWORD;
    DECLARE socket$o structure
            (host$id            WORD,
            port$id             WORD) AT (@socket);
    DECLARE p$control$msg(16)   BYTE;
    DECLARE send$data(100)      BYTE;
    DECLARE data$adr            DWORD;
    DECLARE data$length         WORD;
    DECLARE flags               WORD;
    DECLARE exception           WORD;
    DECLARE slot    LITERALLY   '1H';
    DECLARE port    LITERALLY   '1F4H';
    DECLARE null    LITERALLY   '0H';
    DECLARE length  LITERALLY   '64H';

    CODE: DO;
            socket$o.host$id = slot;
            socket$o.port$id = port;
            data$length      = length;
            flags            = null;

            data$adr = CONVERT_ADDRESS (@send$data(0));

                    .
                    .  (Typical code to define
                    .   the 16-byte p$control$msg block
                    .   holding the control message.)
                    .
                    .  (Typical code to define
                    .   the 100-byte message
                    .   portion.)

            CALL BS$SEND
                (socket,@p$control$msg(0),data$adr,
                data$length,flags,@exception);

            IF exception <> 0
                THEN CALL BSERROR;
                    .
                    .  (Typical code to execute
                    .   for successful status.)
                    .
    END CODE;
END SAMPLE_BS$SEND;
```

CONDITION CODES

E$OK	0000H	No exceptional conditions.
BSETRANSMISSION	00E1H	An error occurred while transmitting a MULTIBUS II message.

CALL BS$RECEIVE (socket,control$ptr,data$adr,data$length, exception$ptr)

INPUT PARAMETERS

socket	A DWORD of the form host$id:port$id identifying the remote sender.
control$ptr	A POINTER to the area in memory that receives the control message.
data$adr	A DWORD containing the absolute address of a data message received. If data$adr is NULL (0), then the host CPU board expects a control message. Otherwise, data$adr points to a contiguous buffer that receives the data portion of the message.
data$length	A WORD defining the length of the data message received.

OUTPUT PARAMETERS

exception$ptr	A POINTER to a WORD to which the Operating System returns the exception code generated by this Bootstrap Loader Communication System call.

DESCRIPTION

The utility BS$RECEIVE enables a host CPU board to receive a message from a specific device controller. The utilities call identifies the MBII slot to wait on, the type of message, and addresses for the control portion and, if necessary, the data portion of the message.

To receive data messages, you must provide a buffer containing adequate space in which to capture the data. If you do not supply a large enough buffer, the receiving CPU host rejects the message. Also, your application must make a call to BS$RECEIVE before the actual message is sent. No facility for queuing asynchronously received messages exist.

EXAMPLE

This example illustrates the fundamentals of message passing from the bootable device controller board to the host CPU board. This example is written in PL/M-86 code and is intended to be generic in nature.

```
/*************************************************************
 *     This example illustrates how a host CPU board         *
 *     receives a data message from the bootable             *
 *     controller board located in slot 1 at port 500 of     *
 *     the MULTIBUS II system.                               *
 *                                                           *
 *     The control portion of the message received is        *
 *     located in the 16-byte array p$status$msg             *
 *     (Peripheral Status Message).  The data portion of     *
 *     the message received is located in the 1024-byte      *
 *     data array sent$data.                                 *
 *************************************************************/

SAMPLE_BS$RECEIVE: DO;

    DECLARE socket              DWORD;
    DECLARE socket$o structure
            (host$id            WORD,
            port$id             WORD) AT (@socket);
    DECLARE p$status$msg(16)    BYTE;
    DECLARE data$adr            DWORD;
    DECLARE data$length         WORD;
    DECLARE flags               WORD;
    DECLARE exception           WORD;
    DECLARE sent$data(1024)     BYTE;
    DECLARE slot    LITERALLY   '1H';
    DECLARE port    LITERALLY   '1F4H';
    DECLARE length  LITERALLY   '400H';
    DECLARE null    LITERALLY   '0H';

    CODE: DO;
            socket$o.host$id = slot;
            socket$o.port$id = port;
            data$length      = length;
            flags            = null;

            data$adr = CONVERT_ADDRESS (@sent$data(0));

            CALL BS$RECEIVE
                (socket,@p$status$msg(0),data$adr,
                data$length,flags,@exception);
```

```
            IF exception <> 0
                THEN CALL BSERROR;
                     .
                     .  (Typical code to execute
                     .   for successful status.)
                     .
    END CODE;
END SAMPLE_BS$RECEIVE;
```

CONDITION CODES

E$OK	0000H	No exceptional conditions.
BSEBUFFER$SIZE	00E2H	The receive data buffer posted is too small.
BSETRANSMISSION	00E1H	An error occurred while transmitting a MULTIBUS II message.

5.5.5 Message Broadcasting

Message broadcasting enables one control message to go out at the same time to all boards (bus agents) in the MULTIBUS II system. Recall that the identification scheme for boards employs sockets, which have the host$id:port$id form. Host$id indicates the board involved and port$id indicates the unique I/O port within the board. During message broadcasting, the host$id portion of the socket is uninterpreted. Thus, the message arrives at every board having a port identified by port$id.

The Bootstrap Loader Communication System uses the bs$broadcast utility to support message broadcasting. The following utility description presents bs$broadcast:

CALL BS$BROADCAST (socket,control$ptr,exception$ptr)

INPUT PARAMETERS

socket	A DWORD of the form host$id:port$id identifying the remote destination. The host$id component is ignored.
control$ptr	A POINTER to the control message sent.

OUTPUT PARAMETER

exception$ptr	A POINTER to a WORD to which the Operating System returns the exception code generated by this Bootstrap Loader Communication System call.

DESCRIPTION

The bs$broadcast utility transmits a single control message to the MULTIBUS II boards having a port whose ID matches the port$id portion of the parameter socket. This message goes out on all MULTIBUS II buses (iPSB parallel system bus and/or iSSB serial system bus) connected to the broadcasting CPU host board.

EXAMPLE

This example illustrates the fundamentals of broadcasting control messages over a MULTIBUS II system. This example is written in PL/M-86 code and is intended to be generic in nature.

```
/*************************************************************
 *     This example illustrates how a host CPU board         *
 *     broadcasts a control message to all system boards     *
 *     having a port$id of 500.  During message              *
 *     broadcasting, the host$id portion of socket is        *
 *     ignored.                                              *
 *                                                           *
 *     The control message sent is located in p$control$msg  *
 *     (Peripheral Command Message.                          *
 *************************************************************/

SAMPLE_BS$BROADCAST: DO;

    DECLARE socket              DWORD;
    DECLARE socket$o structure
            (host$id            WORD,
            port$id             WORD) AT (@socket);
    DECLARE p$control$msg(20)   BYTE;
    DECLARE exception           WORD;
    DECLARE slot    LITERALLY   '1H';
    DECLARE port    LITERALLY   '1F4H';

    CODE: DO;
            socket$o.host$id = slot;
            socket$o.port$id = port;

            CALL BS$BROADCAST
                (socket,@p$control$msg(0),@exception);
```

```
            IF exception <> 0
                THEN CALL BSERROR;
                     .
                     .  (Typical code to execute
                     .   for successful status.)
                     .
    END CODE;
END SAMPLE_BS$BROADCAST;
```

CONDITION CODES

E$OK	0000H	No exceptional conditions.
BSETRANSMISSION	00E1H	An error occurred while transmitting a MULTIBUS II message.

5.5.6 Transmission Modes

Data message transmissions are synchronous in that the sender of the message waits for the receiver of the message to return a transmission status value. This value indicates whether or not the receiver successfully acquired the message. Control messages, however, are not synchronous in this manner. There is no indication to the sender that a control message has been received. Also, related to each type of message transmission is a transaction ID value. The communication system uses this value internally to match requests with responses and to indicate whether the message is an rsvp message or a nonrsvp message. If the message sent is not an rsvp message, the associated transaction ID value is zero. If the message sent is an rsvp message, the associated transaction ID value is a nonzero value matched to both the request and the response.

5.5.7 Interconnect Space

The Bootstrap Loader Communication System supports access to board interconnect space. This access enables the driver to determine critical device status information. The Bootstrap Loader Communication System provides interconnect space access through two system utilities: BSGETINTERCONNECT and BSSETINTERCONNECT. When you use these calls within your driver code, you must verify the value read or written from or to the interconnect space is what you expect. The Bootstrap Loader code does not know what "correct" values should be.

The following utility description presents BSGETINTERCONNECT:

---

value = BSGETINTERCONNECT (slot$number,reg$number, exception$ptr)

INPUT PARAMETERS

slot$number — A BYTE that specifies the MBII slot whose interconnect space is to be read. You must specify this value as follows:

Value	Meaning
0-19	specifies iPSB slot numbers 0-19
20-23	illegal values
24-29	specifies iLBX slot numbers 0-5
30	illegal
31	specifies the iPSB slot of the CPU that the calling software is executing on, regardless of the actual iPSB slot number of the CPU
32-255	illegal values

reg$number — A WORD identifying the interconnect register to be read. This value must be in the range of 0000H to 01FFH (the interconnect space definition).

OUTPUT PARAMETERS

value — A BYTE containing the contents of the interconnect register read.

exception$ptr — A POINTER to a WORD in which the Bootstrap Loader returns the exception code generated by this Bootstrap Loader Communication System call.

DESCRIPTION

The utility BSGETINTERCONNECT reads the contents of the interconnect register specified by reg$number from the board specified by slot$number and returns the contents in the parameter value.

EXAMPLE

This example illustrates the fundamentals of reading interconnect space registers. The example is written in PL/M-86 code and is intended to be generic in nature.

```
/*************************************************************
 *  This example reads the general purpose register of the   *
 *  unit definition record within the interconnect space     *
 *  found on the board in slot number three.  Note that      *
 *  this code does no checking of status after each call to  *
 *  BS$GET$INTERCONNECT.  The programmer must ensure the     *
 *  value returned is correct.                               *
 *************************************************************/

SAMPLE_BS$GET$INTERCONNECT: DO;

    DECLARE slot$number          BYTE;
    DECLARE record$offset        WORD;
    DECLARE unit$def$rec         BYTE;
    DECLARE rec$length$reg$off   BYTE;
    DECLARE gen$status$reg$off   BYTE;
    DECLARE record$found         BYTE;
    DECLARE eot$rec              BYTE;
    DECLARE status               WORD;
    DECLARE value                BYTE;
    DECLARE slot    LITERALLY    '3H';
    DECLARE udr     LITERALLY    '0FEH';
    DECLARE gsro    LITERALLY    '0AH';
    DECLARE eotrec  LITERALLY    '0FFH';
    DECLARE rlro    LITERALLY    '01H';
    DECLARE ro      LITERALLY    '020H';

    CODE: DO;
            slot$number        = slot;
            unit$def$rec       = udr;
            gen$status$reg$off = gsro;
            eot$rec            = eotrec;
            rec$length$reg$off = rlro;

/*************************************************************
 *  Set up to read the first nonheader record within the     *
 *  interconnect space.  This is done by establishing        *
 *  record$offset past the interconnect space header         *
 *  record, which in this case is 32 bytes long.             *
 *************************************************************/

            record$offset = ro;
```

```
/*************************************************************
 *  Read the record type register (the first register       *
 *  within a record) of the first nonheader record into     *
 *  the variable record$found.                              *
 *************************************************************/

          record$found = BS$GET$INTERCONNECT
                         (slot$number,
                         record$offset,
                         @status);

/*************************************************************
 *  Determine if this first record is the record we want to *
 *  read from.  If so, bypass the DO WHILE loop and get     *
 *  right to reading the specific register.  If not,        *
 *  and the record is not the EOT (End Of Template) record, *
 *  execute the DO WHILE loop to get at the next record.    *
 *************************************************************/

          DO WHILE (record$found <> unit$def$rec) AND
                   (record$found <> eot$rec);

/*************************************************************
 *  Position record$offset to read the next sequential      *
 *  record.  This is done by calling BS$GET$INTERCONNECT    *
 *  to read the current record length, adding 2 (for the    *
 *  two bytes used for the record type and record length    *
 *  registers), and finally adding the current              *
 *  record$offset value.  Note that record$offset +         *
 *  rec$length$reg$off yields the interconnect register     *
 *  that holds the current record length.                   *
 *************************************************************/

          record$offset = record$offset + 2 +
              BS$GET$INTERCONNECT
                  (slot$number,
                  record$offset +
                  rec$length$reg$off,
                  @status);

/*************************************************************
 *  Read the next record-type register into the variable    *
 *  record$found.                                           *
 *************************************************************/
```

```
            record$found = BS$GET$INTERCONNECT
                           (slot$number,
                           record$offset,
                           @status);

            END;

/*************************************************************
 *  Call BS$GET$INTERCONNECT to read the general status      *
 *  register.  The exact register location is determined by  *
 *  adding the register offset value gen$status$reg$off to   *
 *  record$offset                                            *
 *************************************************************/

        value = BS$GET$INTERCONNECT(slot$number,
                record$offset + gen$status$reg$off,
                @status);

    END CODE;
END SAMPLE_BS$GET$INTERCONNECT;
```

CONDITION CODES

E$OK	0000H	No exceptional conditions.

The following utility description presents BSSETINTERCONNECT:

CALL = BSSETINTERCONNECT	(value,slot$number,reg$number, exception$ptr)

INPUT PARAMETERS

value A BYTE containing the value to be written into the interconnect register.

slot$number — A BYTE specifying the MBII slot whose interconnect space is to be written. You must specify this value as follows:

Value	Meaning
0-19	specifies iPSB slot numbers 0-19
20-23	illegal values
24-29	specifies iLBX slot numbers 0-5
30	illegal
31	specifies the iPSB slot of the CPU that the calling software is executing on, regardless of the actual iPSB slot number of the CPU
32-255	illegal values

reg$number — A WORD identifying the interconnect register to be written. This value must be in the range of 0000H to 01FFH (the interconnect space definition).

OUTPUT PARAMETERS

exception$ptr — A POINTER to a WORD in which the Bootstrap Loader returns the exception code generated by this Bootstrap Loader Communication System call.

DESCRIPTION

The utility BSSETINTERCONNECT writes the interconnect register specified by reg$number on the board specified by slot$number with the contents in the parameter value.

EXAMPLE

This example illustrates the fundamentals of writing interconnect space registers. The example is written in PL/M-86 code and is intended to be generic in nature.

```
/*************************************************************
 *  This example writes the controller initialization       *
 *  register of the parallel system bus control record      *
 *  within the interconnect space found on the board in     *
 *  slot number three.  Note that this code does no         *
 *  checking of status after each call to                   *
 *  BS$GET$INTERCONNECT and BS$SET$INTERCONNECT.  The       *
 *  programmer must ensure values returned and written are  *
 *  correct.                                                *
 *                                                          *
 *  This example uses the same record-searching scheme      *
 *  shown in the example for BS$GET$INTERCONNECT.           *
 *************************************************************/

SAMPLE_BS$SET$INTERCONNECT: DO;

    DECLARE slot$number          BYTE;
    DECLARE status               WORD;
    DECLARE record$offset        WORD;
    DECLARE psb$ctrl$rec         BYTE;
    DECLARE rec$length$reg$off   BYTE;
    DECLARE contr$init$reg$off   BYTE;
    DECLARE record$found         BYTE;
    DECLARE eot$rec              BYTE;
    DECLARE host$mess$id         BYTE;
    DECLARE slot    LITERALLY    '3H';
    DECLARE psbcr   LITERALLY    '6H';
    DECLARE ciro    LITERALLY    'DH';
    DECLARE eotrec  LITERALLY    '0FFH';
    DECLARE rlro    LITERALLY    '01H';
    DECLARE hmid    LITERALLY    'AH';
    DECLARE ro      LITERALLY    '020H';

    CODE: DO;
            slot$number         = slot;
            psb$ctrl$rec        = psbcr;
            contr$init$reg$off  = ciro;
            eot$rec             = eotrec;
            rec$length$reg$off  = rlro;
            host$mess$id        = hmid;
```

```
/************************************************************
 *  Set up to read the first nonheader record within the    *
 *  interconnect space.  This is done by establishing       *
 *  record$offset past the interconnect space header        *
 *  record, which in this case is 32 bytes long.            *
 ************************************************************/

            record$offset = ro;

/************************************************************
 *  Read the record type register (the first register       *
 *  within a record) of the first nonheader record into     *
 *  the variable record$found.                              *
 ************************************************************/

            record$found = BS$GET$INTERCONNECT
                           (slot$number,
                           record$offset,
                           @status);

/************************************************************
 *  Determine if this first record is the record we want to *
 *  write.  If so, bypass the DO WHILE loop and proceed     *
 *  writing the specific register.  If not, and the record  *
 *  is not the EOT (End Of Template) record, execute the DO *
 *  WHILE loop to get at the next record.                   *
 ************************************************************/

            DO WHILE (record$found <> psb$ctrl$rec) AND
                     (record$found <> eot$rec);

/************************************************************
 *  Position record$offset to read the next sequential      *
 *  record.  This is done by calling BS$GET$INTERCONNECT    *
 *  to read the current record length, adding 2 (for the    *
 *  two bytes used for the record type and record length    *
 *  registers), and finally adding the current              *
 *  record$offset value.  Note that record$offset +         *
 *  rec$length$reg$off yields the interconnect register     *
 *  that holds the current record length.                   *
 ************************************************************/

            record$offset = record$offset + 2 +
                BS$GET$INTERCONNECT
                    (slot$number,
                    record$offset +
                    rec$length$reg$off,
                    @status);
```

```
/*************************************************************
 *  Read the next record-type register into the variable     *
 *  record$found.                                            *
 *************************************************************/

            record$found = BS$GET$INTERCONNECT
                           (slot$number,
                           record$offset,
                           @status);

        END;

/*************************************************************
 *  Call BS$SET$INTERCONNECT to write the controller         *
 *  initialization register.  The exact register location    *
 *  is determined by adding the register offset value        *
 *  contr$init$reg$off to record$offset.                     *
 *************************************************************/

            CALL BS$SET$INTERCONNECT(host$mess$id, slot$number,
                record$offset + contr$init$reg$off, @status);

    END CODE;
END SAMPLE_BS$SET$INTERCONNECT;
```

CONDITION CODES

E$OK	0000H	No exceptional conditions.

5.5.8 Driver Code Considerations

When writing the first-stage driver, you must provide two procedures to the Bootstrap Loader: a device initialization procedure and a device read procedure. To be compatible with the Bootstrap Loader, these procedures must perform the same steps as the initialization and read procedures listed in Sections 5.2 and 5.3.

An additional requirement for driver code used in a MULTIBUS II environment stipulates that code using any of the utilities shown in Sections 5.5.2 through 5.5.6 belong to the Bootstrap Loader Drivers COMPACT sub-system. The reason for this requirement is because all the utilities are accessible as NEAR calls.

The following partial code provides an example of how to ensure your driver code is part of the Bootstrap Loader Driver COMPACT sub-system. In this example, the coding is shown using the ASM86 programming language.

```
name    bs2pci

public device_init_224A
public device_read_224A

bsl_drivers_cgroup      group   bsl_drivers_code
bsl_drivers_dgroup      group   bsl_drivers_data

assume cs:  bsl_drivers_cgroup
assume ds:  bsl_drivers_dgroup

bsl_drivers_data    segment word    public  'DATA'
                    .
                    .  (Typical code)
                    .
bsl_drivers_data    ends

bsl_drivers_code    segment byte    public  'CODE'
device_init_224A    proc            far
                    .
                    .  (Typical code)
                    .
bsl_drivers_code    ends
```

In the above example, bs2pci is the name of the driver module. You can name your driver module any unique name you desire.

The two following public statements declare the device initialization and device read procedures as public. These public statements enable the Bootstrap Loader code to access them as FAR calls. Again, you can name your device initialization and read procedures any unique name you desire.

Next, the two group statements ensure that this driver module is grouped together with the Bootstrap Loader utilities as part of the same COMPACT sub-system. You must use the two group names bsl_drivers_cgroup and bsl_drivers_dgroup and the two segment names bsl_drivers_code and bsl_drivers_data.

Finally, the two assume statements establish the correct values for the code segment base address and the data segment base address, cs and ds.

The following algorithm is an example that illustrates both a method of using the Bootstrap Loader Communication System as a way of verifying a certain board is present in the system and of using the utility BSGETINTERCONNECT. The example is written using a pseudo code that is not meant to represent any known programming language.

```
*BEGIN COMMENTS:
*
* Parameters received are BOARD$ID and INSTANCE.
*
*   BOARD$ID is the identification value
*   of the board being looked for.
*
*   INSTANCE is the instance of a particular
*   board on the parallel bus system.  This
*   parameter allows for multiple occurrences of
*   the same board within the MULTIBUS II system.
*
* Parameters returned are iPSB$SLOT
*
*   iPSB$SLOT is the MULTIBUS II board slot when the board
*   is found, or the value 0FFH when the board is
*   not found.
*
*   Note that the variable VENDOR_ID points to the
*   specific interconnect space register that
*   contains the board identification value.
*
*END COMMENTS:
************************************************
*
*BEGIN CODE:
*
* DO until all MULTIBUS II board slots on the PSB are
*    sequentially examined.  Use the variable
*    iPSB$SLOT as the looping variable to indicate
*    the slot number for the board being examined.
*
*    VENDOR$ID = BS$GET$INTERCONNECT(iPSB$SLOT,
*                    VENDOR_ID, STATUS)
```

```
*
*   If
*    the VENDOR$ID returned is nonzero,
*    a board exists in the examined slot
*   then
*      If
*       VENDOR$ID matches BOARD$ID
*      then
*         If
*          INSTANCE is the desired instance of
*          BOARD$ID
*         then
*            return the iPSB$SLOT looping index to
*            indicate the slot number of BOARD$ID
*         else
*      else
*   else
*     If we have checked all board slots
*     then
*        Return the value 0FFH as the iPSB$SLOT
*        parameter indicating the device
*        to boot from does not exist.
*     else
*        Loop back to beginning to check the next
*        board slot.
*
* END DO
*
*END CODE:
```

5.6. CHANGING BS1.A86 OR BS1MB2.A86 TO INCLUDE THE NEW FIRST-STAGE DRIVER

The first stage of the Bootstrap Loader obtains information about the devices and their associated device drivers from the Bootstrap Loader configuration file BS1.A86 or BS1MB2.A86. To support a custom device driver, you must add to that file a %DEVICE macro for each unit on the device that your first-stage device driver supports. For example, if two flexible diskette drives are attached to the device, you must add two %DEVICE macros to the list (one for each drive). Chapter 3 describes the syntax of the %DEVICE macro.

As an example, Figure 5-3 shows a portion of the BS1.A86 file that was changed to add %DEVICE macros for two units supported by a custom first-stage driver (changes to BS1MB2.A86 would occur similarly). The units have numbers 0 and 1, and their physical names are YZ0 and YZ1, respectively. The name of the custom driver device initialization procedure is NEWDEVICEINIT, and the name of the device read procedure is NEWDEVICEREAD. Arrows to the left of the figure show the added lines.

```
    name    bs1

    $include(:f1:bs1.inc)

    %cpu(80286)

    ;iSBC 188/48 initialization of the iAPX 188
    ;iAPX_186_INIT(y,0fc38h,none,80bbh,none,003bh)
            .
            .
            .
    %device(b0, 0, deviceinit254, deviceread254)
    %device(ba0, 0, deviceinit264, deviceread264)
--> %device(yz0, 0, newdeviceinit, newdeviceread)
--> %device(yz1, 1, newdeviceinit, newdeviceread)
    %end
```

Figure 5-3. Modified BS1.A86 File

5.7. GENERATING A NEW FIRST STAGE CONTAINING THE CUSTOM DEVICE DRIVER

Once you have written the custom device driver and changed the Bootstrap Loader Configuration files, you must generate a new first stage that includes the custom device driver. To do so, follow the steps below. (These steps assume that you use an iRMX II system to develop your code.)

1. Compile or assemble the first-stage device initialization and device read procedures. For example, the following command assembles device read and device initialize procedures that are assumed to reside in the file NEWDEVICE1.A86.

```
  - asm86 newdriv1.a86 object(newdriv1.obj)
iRMX II 8086/87/186 MACRO ASSEMBLER, V2.0
Copyright 1980, 1981, 1982, INTEL CORP.
ASSEMBLY COMPLETED, NO ERRORS FOUND
```

2. Insert the object modules for the device read and the device initialize procedures into the object library of the Bootstrap Loader. This library is named BS1.LIB and normally resides in the directory /RMX286/BOOT or /RMX86/BOOT. The following commands add the object modules generated in Step 1.

```
  - LIB86
iRMX II 8086 LIBRARIAN V2.0
Copyright 1980 INTEL CORPORATION
*add newdrivl.obj to /rmx286/boot/bs1.lib
*
```

3. Attach the directory containing the Bootstrap Loader configuration files as the current default directory:

```
  - attachfile /rmx286/boot
/rmx286/boot, attached AS :$:
```

4. Generate a new first stage by invoking the SUBMIT file named BS1.CSD. Chapter 2 describes the details of the invocation. As an example, the following command assumes that you have chosen 40000H as the memory location of the first stage and 43000H as the memory location of the second stage.

   -

   This step assumes that you have made appropriate changes to the BS1.CSD file as described earlier in this chapter.

   The BS1.CSD file places the resulting located Bootstrap Loader in the file BS1.

One thing to remember about this procedure is that because you added your device driver to the object library of the Bootstrap Loader, the device driver is automatically included in all future versions of the first stage created by BS1.CSD.

CHAPTER 6
WRITING A CUSTOM THIRD-STAGE DRIVER

6.1 INTRODUCTION

If you plan to use the Bootstrap Loader to load iRMX II applications from a device for which no Intel-supplied third-stage driver exists, you can make one of two choices dependent upon the size of your loadfile:

- For loadfiles smaller than 840K bytes, use the generic third stage. The generic third stage uses the first-stage device drivers you have already supplied. Since the loadfile fits in the 1 megabyte address space supported in real mode, and first-stage device drivers are able to place the loadfile, no need for you to create new device drivers exists for the third stage.
- For loadfiles larger than 840K bytes, use the device-specific third stage. The device-specific third stage uses new device drivers that you must supply. These device drivers run in protected virtual address mode enabling the loadfile to be placed using the full 16 megabyte range of addresses.

This chapter outlines the procedure for writing a third-stage driver needed for the device-specific third stage. To assist you in writing your own drivers, the iRMX II package contains the source code for a working third stage driver. After installing your iRMX II system, you can find the source code in the file /RMX286/BOOT/BPMSC.A86.

6.2 WHAT A THIRD-STAGE DEVICE DRIVER MUST CONTAIN

The third stage device driver, like the first stage, must contain a device initialization and a device read procedure. For the most part, these procedures are similar to their first-stage counterparts. However, two differences exist.

- Both procedures must reside in the same code segment.
- You must also create a PUBLIC symbol that contains a pointer to the device driver data segment. The third stage needs this information so that it can create a descriptor for the data segment, enabling the driver to access the segment in protected mode.

When developing code for your third stage driver, you must remember that the second stage always loads the third stage, including the drivers you write. The only type of code that the second stage can load is code that uses the 8086 object module format (OMF-86). Therefore, you must use 8086 tools (ASM86, PL/M-86, LINK86, etc.) to develop the third-stage device initialization and read procedures.

Even though you use 8086 tools to develop your driver code, the resulting initialization and read procedures must be able to run in protected mode. One ramification of running in protected mode is that all long pointers produced by PL/M-86 (or by any other means) that were correct in real mode cause an ILLEGAL SELECTOR exception in protected mode. Therefore, if you must use long pointers, your device initialization and read procedure must determine whether or not the processor is in protected mode. If protected mode is active, the procedure must replace all the selector portions of long pointers with a new selector that is valid in protected mode.

You can determine the processor mode by using the following assembly code:

```
DB 0FH,01H,0E3H      ;Opcode for the ASM286 instruction
                     ;SMSW BX.  You must use
                     ;DB 0FH,01H,0E3H because SMSW is an
                     ;ASM286 instruction unrecognized by
                     ;ASM86.
AND BX, 01H          ;Examine lowest bit of MSW to see if
                     ; CPU is running in PVAM.
JZ REAL              ;No, not running in PVAM.
.                    ;
. code to override   ;Yes, running in PVAM.
. selectors of       ;
. long pointers      ;
.                    ;
```

If your driver code is going to operate in the MULTIBUS II environment, two additional driver code constraints exist. First, you must follow the MULTIBUS II transport protocol for communication between the driver and the device controller you bootstrap load from. You can accomplish this by using Bootstrap Loader Communication System utility calls within your driver code. Second, you must organize your driver code so that it belongs to the BSL-Drivers COMPACT sub-system. This last requirement is necessary because the Bootstrap Loader Communication System utilities are all NEAR calls.

The next two sections describe the interface these procedures must present to the third stage. The sections after that describe how to supply configuration information to the driver and how to generate a third stage that includes the new driver.

6.3 DEVICE INITIALIZATION PROCEDURE

The device initialization procedure must present the following PL/M-86 interface to the third stage:

```
device$init: PROCEDURE (unit) WORD PUBLIC;
DECLARE unit WORD;
     .
     .
     .
END device$init;
```

where:

device$init	The name of the device initialization procedure. You can choose any name you wish for this procedure, as long as it does not conflict with the names of other third-stage procedures.
unit	The device 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 third-stage device driver initialization procedure, (like the first-stage device initialization procedure) must perform the following operations:

1. Test to see if the device is present. If the device is not present, 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. Read the device volume label to to obtain the device granularity. (For information on the location and organization of the volume label, see the *iRMX 86 Disk Verification Utility manual.*)
5. If the attempt to obtain the device granularity was successful, return the device granularity. Otherwise, return the value zero.

NOTE

In addition to the above five steps, the procedure must follow MULTIBUS II transport protocol and belong to the BSL-Drivers COMPACT sub-system if the driver functions in a MULTIBUS II environment. Refer to Section 5.5 for more information on these two requirements.

Notice that the functions of the first-stage and the third-stage device initialization procedures are identical. Therefore, you can take two courses of action to provide a device initialization procedure for the third-stage custom driver.

1. You can allow the first-stage custom driver and the third-stage custom driver to share the same data segment. In this case, the third-stage device initialization procedure is redundant because the device was initialized by the first stage and any data in the data segment remains valid.

   Because the third stage calls the device initialization procedure regardless of your intentions, you must supply a third-stage driver device initialization procedure even if it is redundant. However, the device initialization procedure can be an empty routine whose only function is to return the device granularity read from the common data segment.

2. You can require the first-stage and third-stage drivers to use different data segments. In this case, the first-stage and third-stage initialization procedures must independently initialize their respective data segments. With this arrangement, you must provide two complete device initialization routines. However, because their functions are identical (except for assigning a value for the data segment), you can use the same code for both procedures.

6.4 DEVICE READ PROCEDURE

The device read procedure must present the following PL/M-86 interface to the third stage:

```
device$read: PROCEDURE (unit, blk$num, buf$ptr) PUBLIC;
DECLARE unit WORD;
DECLARE blk$num DWORD;
DECLARE buf$ptr POINTER;
    .
    . (code)
    .
END device$read;
```

where:

device$read	The name of the device read procedure. You can choose any name you wish for this procedure, as long as it does not conflict with the names of any other third-stage procedure.
unit	The device unit number as specified during Bootstrap Loader configuration.

blk$num	A 32-bit value specifying the number of the block that the Bootstrap Loader wants the procedure to read. Each block is of device granularity size, with the first block on the device being block 0.
buf$ptr	A 32-bit pointer to the buffer in which the device read procedure must copy the information it reads from the secondary storage device.

The device read procedure does not return a value to the caller. It simply reads data from the bootstrap device and places it in the memory location specified by the buf$ptr parameter.

The third-stage and first-stage device read procedures perform similar functions. Therefore, you may want to create the third-stage read procedure by performing modifications on the first-stage read procedure (if, for instance, it has already been written and resides in PROM). If the first-stage read procedure does not yet exist, you can write the third-stage read procedure first and then modify it to create the first-stage procedure.

Unlike the Bootstrap Loader first stage, the third stage has no built-in facilities for reporting I/O errors. That is, the device driver cannot call BS$ERROR. Therefore, if you require I/O error reporting, you must write a complete custom error-checking mechanism and include it in the device read procedure. (For an explanation of BS$ERROR, refer to Chapter 3.)

To be compatible with the Bootstrap Loader, the device read procedure must perform the following steps:

1. Save the third stage DS (the data segment selector of the calling routine), and then copy the driver data segment selector from the AX register into the DS register. (When calling the device read procedure, the third stage puts the driver data segment selector in the AX register.) The device read procedure must perform this function immediately.

   Because register manipulation is not possible with high-level languages (such as PL/M-86), you must write this portion of the device read procedure in assembly language (ASM86).

2. Check whether the processor is in real or protected mode. If the processor is in protected mode, you may want to initialize other selectors to appropriate values (buf$ptr for example). Assuming Step 1 has already been accomplished, you need not initialize the code (CS), data (DS), and stack (SS) registers. These registers will already be set correctly.

3. Read the block (specified by the blk$num parameter) from the bootstrap device (specified by the unit parameter) and place the data in the memory location specified by the buf$ptr parameter.

4. Restore the third stage data segment selector to the DS register. As with Step 1, you must write this code in assembly language, because it involves register manipulation.

NOTE

In addition to the above steps, the procedure must follow MULTIBUS II transport protocol and belong to the BSL-Drivers COMPACT sub-system if the driver functions in a MULTIBUS II environment. Refer to Section 5.5 for more information on these two requirements.

6.5 PROTECTED MODE CONSIDERATIONS

Because you develop your driver procedures using 8086 tools and run the procedures in protected mode, you should keep several items in mind:

- When the third stage calls the device read procedure, it puts the driver data segment selector in the AX register. When first called, the device read procedure must save the DS used by the caller (the third stage data segment selector), and then copy the driver data segment selector from the AX register into the DS register. Before exiting, the procedure must restore the original contents of the DS register. If you are writing in assembly language, you can perform this operation as follows:

```
THE$DEVICE$READ PROC FAR
    PUSH BP             ;Get Addressability to
                        ;arguments
    MOV BP, SP
    PUSH DS             ;Save third stage DS
    MOV DS, AX          ;Get local data segment
    .
    .                   ;Perform the device read
    .                   ;functions
    .
    POP DS              ;Restore third stage DS
    POP BP              ;Restore BP
    RET 8               ;Return
THE$DEVICE$READ ENDP
```

If you are writing code in a high-level language (such as PL/M-86), you still must code this function in assembly language. The reason for this restriction is because higher level languages do not allow you to manipulate registers directly. You can, however, combine assembly language with your high-level language by writing an assembly language "shell" that handles the register manipulation and then calls a PL/M-86 procedure to perform the other device read functions. For instance, the following example saves the third stage DS, calls a high-level language routine to do the device read, and restores the third stage DS register before returning.

```
THE$DEVICE$READ PROC FAR
    PUSH BP             ;Get Addressability to
                        ;arguments
    MOV BP, SP
    PUSH DS             ;Save third stage DS
    MOV DS, AX          ;Get local data segment
    .
    CALL PLMREAD        ;Call a PLM procedure to
                        ;perform the
                        ;device functions
    .
    POP DS              ;Restore third stage DS
    POP BP              ;Restore BP
    RET 8               ;Return
THE$DEVICE$READ ENDP
```

- Be careful when changing DS, SS, CS, or ES registers while in protected mode. They point to valid entries in the global descriptor table (GDT) that were prepared for your driver by the third stage. If you change any of these registers, the new value must be a valid GDT entry or an ILLEGAL SELECTOR or a GENERAL PROTECTION exception will occur.
- Do not link your code to PLM86.LIB, because the compiler issues long calls to procedures in that library. These long calls cause exceptions when the calls are attempted in protected mode.
- The buf$ptr parameter the third stage passes to the device read procedure is a valid pointer in real mode only. You can pass this value to the device as a physical address, but do not try to use it as a pointer in protected mode. If you require a pointer, replace the buff$ptr selector with the third stage DS value. This DS value is intact when the device read procedure is called.

6.6 SUPPLYING CONFIGURATION INFORMATION TO THE THIRD-STAGE DRIVER

Like a first-stage device driver, all third-stage drivers require configuration information about the devices they support. You can provide this information either by hard-coding it into the driver or by creating a special configuration file for the device. Both of these techniques are the same for the first and third stages. Refer to the section in Chapter 5 entitled "Supplying Configuration Information to the First Stage" for descriptions of these techniques.

If you decide to create configuration files for your first-stage and third-stage drivers, you should probably use a single configuration file for each device and link it to both the first-stage and third-stage drivers. The device-specific information is the same for both drivers, and keeping the information in a single file prevents you from giving conflicting information to the two drivers. You can include the configuration file by editing BS3.CSD to assemble and link the configuration file to the third stage. Refer to Section 5.4.2 for an example that shows the similar first-stage process.

6.7 USING MULTIBUS® II TRANSPORT PROTOCOL

If the driver you are creating functions within a MULTIBUS I environment, you need not read this section. Skip to Section 6.8.

If the driver you are creating functions within a MULTIBUS II environment, you must write the driver code to use the MULTIBUS II message transport protocol. To help you accomplish this task, Intel provides a small, single-thread communication system that enables Bootstrap Loader drivers to communicate with device controllers within a MULTIBUS II environment. This system is called the Bootstrap Loader Communication System, and is a subset of the Nucleus Communication System.

Concerning adherence to the MULTIBUS II transport protocol, requirements for third-stage device drivers and first-stage device drivers are identical. Thus, you should refer to Section 5.5 for an overview of the Bootstrap Loader Communication System, the available Bootstrap Loader Communication System utilities, and guidance in writing the device initialization and device read procedures.

Should you desire a more complete description of Bootstrap Loader Communication System concepts similar to Nucleus Communication System concepts, refer to the *Extended iRMX II Nucleus User's Guide* in Volume 2 of the iRMX II documentation set.

6.8 CHANGING BS3.A86 TO INCLUDE THE NEW THIRD-STAGE DRIVER

The device-specific third stage obtains information about the device and its associated device driver from the Bootstrap Loader configuration file BS3.A86. To support a custom device driver, you must add to that file a %DEVICE macro for each unit on the device that your first-stage device driver supports. For example, if two flexible diskette drives are attached to the device, you must add two %DEVICE macros to the list (one for each drive). Chapter 4 describes the syntax of the %DEVICE macro.

Figure 6-1 shows a portion of the BS3.A86 file that was changed to add %DEVICE macros for two units supported by a custom third-stage driver. The arrows in the figure indicate the changes. The new units have numbers 0 and 1, and their physical names are YZ0 and YZ1, respectively. (These physical names must match the names used in the %DEVICE macros in the first-stage configuration file BS1.A86 or BS1MB2.A86.) The name of the custom driver device initialization procedure is NEWDEVICEINIT, and the name of the device read procedure is NEWDEVICEREAD. The public name of the driver data segment is DATA NEWDEV.

```
    $include (:f1:bs3cnf.inc)
    ;
    %device (0,w0,deviceinitmscgen,devicereadmscgen,data_msc)
    %device (1,w1,deviceinitmscgen,devicereadmscgen,data_msc)
    %device (8,wf0,deviceinitmscgen,devicereadmscgen,data_msc)
    %device (9,wf1,deviceinitmscgen,devicereadmscgen,data_msc)
    %device (0,ba0,deviceinit264,deviceread264,data_264)
--> %device(0, yz0, newdeviceinit, newdeviceread, data_newdev)
--> %device(1, yz1, newdeviceinit, newdeviceread, data_newdev)
    ;
    ;int1
    %int3
    ;halt
    ;
    %cpu_board (286/12)
    ;
    %end
```

Figure 6-1. Changing the BS3.A86 File

6.9 GENERATING A NEW THIRD STAGE CONTAINING THE CUSTOM DRIVER

Once you have written the custom device driver and changed the Bootstrap Loader Configuration files, you must generate a device-specific third stage that includes the custom device driver. To do so, perform the following steps. (These steps assume that you use an iRMX system to develop your code.)

1. Compile or assemble the third-stage device initialization and device read procedures. For example, the following command assembles device read and device initialization procedures that reside in the file NEWDRIV3.A86.

```
- asm86 newdriv3.a86 object(newdriv3.obj)
iRMX II 8086/87/186 MACRO ASSEMBLER, V2.0
Copyright 1980, 1981, 1982, INTEL CORP.
ASSEMBLY COMPLETED, NO ERRORS FOUND
```

2. Insert the object modules for the device read and the device initialize procedures into the Bootstrap Loader object library. This library is named BS3.LIB and normally resides in the directory /RMX86/BOOT or /RMX286/BOOT. The following commands add the object modules generated in Step 1.

```
- LIB86
iRMX II 8086 LIBRARIAN V2.0
Copyright 1980 INTEL CORPORATION
*add newdriv3.obj to /rmx286/boot/bs3.lib
*
```

3. Attach the directory containing the Bootstrap Loader configuration files as the current default directory:

```
- attachfile /rmx286/boot
/rmx286/boot, attached AS :$:
```

4. Generate a new third stage by invoking the SUBMIT file named BS3.CSD. Chapter 3 describes the details of invoking BS3.CSD. As an example, the following command names the new third stage "NEW3STG," and locates it at memory location 0BC000H.

```
-
```

This step assumes that you have made any appropriate changes to the BS3.CSD file that are required to support any configuration files you might have designed.

Intel®

CHAPTER 7
ERROR HANDLING

7.1 INTRODUCTION

If the bootstrap loading process is unsuccessful, the Bootstrap Loader initiates error-handling procedures. Notification of failures occurring during the loading process depends on the configuration of the first and third stages. This chapter describes the Bootstrap Loader's error handling facilities.

7.2 ANALYZING BOOTSTRAP LOADER FAILURES

The Bootstrap Loader can display messages at the terminal when bootstrap loading is unsuccessful. As discussed in Chapter 3, the %CONSOLE, %TEXT, and %LIST macros in the BSERR.A86 file determine whether or not messages are displayed when errors occur during the first and second stages, how detailed the messages are, and under what circumstances they are displayed. As Chapter 4 explains, the third stage automatically determines if a monitor is present, and if so, displays error messages at the terminal regardless of the first stage configuration.

The following sections describe what happens when a bootstrap loading error occurs and how to analyze the error. There are two situations described: error analysis when messages are displayed, and error analysis when no messages are displayed.

7.2.1 Actions Taken by the Bootstrap Loader After an Error

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. If the error is detected in the first or second stage, the action taken depends on whether your BSERR.A86 file contains an %AGAIN, %INT1, %INT3, or %HALT macro. If the error is detected in the third stage, the action taken depends on whether your BS3.A86 or BG3.A86 file contains an %INT1, %INT3, or %HALT macro.

The only difference between the device-specific and generic third stages is that the generic third stage never generates the error code "Device Not Supported" (refer to error code 34 later in this chapter), because the generic third stage supports all the devices supported by the first stage. If you invoke the Bootstrap Loader with a device name that is not supported by the first stage, the generic third stage will never even get loaded into memory.

7.2.2 Analyzing Errors 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 the 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 is a short description of the error. Although the second part is always displayed for third stage errors, it is displayed for first and second stage errors only if the %TEXT or %LIST macro is included.

Each numerical error code has two digits. The first digit indicates, if possible, the stage of the bootstrap loading process in which the error occurred. The second digit distinguishes the types of errors that can occur in a particular stage. There are four possible values for the first digit.

First Digit	Stage
0	Can't tell
1	First
2	Second
3	Third

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.

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, so that you can further diagnose the problem. This byte identifies the driver for the device that produced the error, as follows:

Code	Driver
08H	208
15H	MSC (with or without 218A)
18H	218A on CPU board
25H	186/224A
51H	251
54H	254 or 264
0E0H	SCSI
other (in range A0H-DFH)	driver for your custom

Note that this device code is overwritten when the description is printed if the %TEXT or %LIST macro is 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 5.

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 only one device is 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 has no 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 enter 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 could not 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. In iRMX II systems, the Bootstrap Loader could not find the third stage.

Error Code: 22
Description: Bad checksum.

While trying to load the target file (the application system for iRMX I systems, or the third stage for iRMX II systems), the Bootstrap Loader encountered a checksum error.

Each file consists of several records. 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 target file (the application system for iRMX I systems, or the third stage for iRMX II systems).

Error Code: 24
Description: No start address found in input file.

The Bootstrap Loader successfully loaded the target file but was unable to transfer control to the file, because initial CS and IP values were not present.

Error Code: 31
Description: File not found.

The third stage was unable to find the target file on the designated bootstrap device. Regardless of the way you invoked the Bootstrap Loader, the target file is expected to have a .286 extension.

Error Code: 32
Description: Bad checksum.

The third stage encountered a checksum error while trying to load the target file.

Error Code: 33
Description: Premature end of file.

The third stage reached end-of-file earlier than expected while attempting to load the target file.

Error Code: 34
Description: Device not supported.

The specified device is not supported by the device-specific third stage. That is, there is no %DEVICE macro invocation for this device in the BS3.A86 file.

Error Code: 35
Description: Invalid file type.

The target file is not an 80286 bootloadable file (usually produced by BLD286).

7.2.3 Analyzing Errors Without Displayed Error Messages

In most cases, you can determine the cause of a Bootstrap Loader failure by observing the behavior of the Bootstrap Loader when it fails to load the application successfully. You can then take steps to correct the failure. Table 4-1 lists some common behaviors and possible causes for 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, the effects of which are 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, ensure that neither the Bootstrap Loader nor the target file overlaps the device controller block for the device.

Table 7-1. 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 has no end-of-file record or no start address. The file contains a checksum error. An I/O error is occurring during the second stage operation.
Bootstrap Loader enters second stage, but does not halt or pass control to the file it loads.	The Bootstrap Loader is attempting to load the system, or third stage, on top of the second stage. The second stage is attempting to load the file into nonexistent memory.
Bootstrap loading fails in the third stage.	The designated file with a .286 extension was not found on the device. The third stage reached an end-of-file earlier than expected. The file contained a checksum error. An I/O error occurred during the third stage operation. The Bootstrap Loader is attempting to load the second stage on top of the Protected Mode third stage.

7.2.4 Initialization Errors

If an error occurs during the initialization of one of the layers of the iRMX I or II Operating System, an error message will be displayed at the console. The message lists the name of the layer whose initialization failed, and gives the iRMX exceptional condition code that indicates the cause of the failure. The following is an example of the kind of message that will be displayed:

```
HI INITIALIZATION:  0021H

Interrupt 3 at 0280:54D8
```

The messages you see will be similar to this one.

Refer to the *Operator's Guide to the iRMX 86 Human Interface* or the *Operator's Guide to the Extended iRMX II Human Interface* for more information about the condition codes.

APPENDIX A
AUTOMATIC BOOT DEVICE RECOGNITION

A.1 INTRODUCTION

Automatic Boot Device Recognition (ABDR) allows the iRMX I or iRMX II 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 ABDR feature in detail. It consolidates information found in other iRMX I 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?

A.2 HOW AUTOMATIC BOOT DEVICE RECOGNITION WORKS

The Nucleus, the Extended I/O System, and the Bootstrap Loader combine to provide the Automatic Boot Device Recognition feature, as follows:

1. 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 the device from which the system was loaded. The name it uses is the one supplied as a parameter in the %DEVICE macro when the Bootstrap Loader was configured.
2. The Bootstrap Loader 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. If CX contains 1234H and DX contains 1235H, the iRMX root job will execute an INT3 instruction before any other code in the Operating System is executed.

4. The Nucleus checks the Root Job's Boolean variable and, if true (equal to 0FFH), places the device name in a segment and catalogues that segment in the root job's object directory under the name RQBOOTED. If it is false (equal to 0), nothing is catalogued in the Root Job's directory. The absence of RQBOOTED from the Root Job's directory indicates the system was not bootloaded or that ABDR was not selected.
5. The Extended I/O System looks up the name RQBOOTED and, if successful, obtains the device name from the segment catalogued there. If the name RQBOOTED is not catalogued in the root directory, the Extended I/O System uses a default device name 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).

A.3 HOW TO INCLUDE AUTOMATIC BOOT DEVICE RECOGNITION

This section describes the operations you must perform to include the ABDR feature in your application. The operations include

- The ABR prompt on the "EIOS" screen (Figure A-1) affects whether the ABDR feature will be included in your application. If you set ABR to "no," the Extended I/O System does not attach a system device. If you set ABR to "yes," the Extended I/O System automatically attaches the system device. The ICU displays another screen (shown in Figure A-2) that lets you specify the characteristics of the system device.

```
   EIOS
-->(ABR) Automatic Boot Device Recognition [Yes/No]  Yes
   (IBS) Internal Buffer Size [0-0FFFFh]             0400H
   (DDS) Default IO Job Directory Size [5-3840]      50
   (ITP) Internal EIOS Task's Priorities [0-255]     131
   (PMI) EIOS Pool Minimum [0-0FFFFH]                    0180H
   (PMA) EIOS Pool Maximum [0-0FFFFH]                    0FFFFFH
   (CD) Configuration directory [1-45 characters]   :SD:RMX286/CONFIG
```

Figure A-1. EIOS Configuration Screen (ABR)

- If you set ABR to "yes," the ICU displays the screen shown in Figure A-2. On this screen, you must specify the characteristics of the system device via the DLN, DPN, DFD, and DO prompts. For the DLN, DFD, and DO prompts, you must not supply this information later in the "Logical Names" screen.

  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 catalogued 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 iSDM 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 D0 prompts, you set other characteristics associated with the system device. For most cases, the defaults (DFD = Named and DO = 0000H) are the preferred values.

```
(ABDR)      Automatic Boot Device Recognition
--->(DLN) Default System Device Logical Name [1-12 chars]   SD
--->(DPN) Default System Device Physical Name [1-12 chars]  w0
--->(DFD) Default System Device File Driver [P/S/N/R]       Named
--->(DO)  Default System Device Owners ID [0-0FFFFH]        0000H
```

Figure A-2. ABDR Screen (DLN, DPN, DFD, DO)

- 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-3 for an example of these prompts.) To enable the ABDR feature to work correctly, you must assign device-unit names and unit numbers that match the device names and unit numbers assigned during Bootstrap Loader configuration.
- 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 BS1MB2.A86). 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 ABDR feature, you must format your system device using the FORMAT command. The *Guide to the Extended iRMX II Interactive Configuration Utility* describes how to set up your system device for use with the current release.

```
    (IMSC) Mass Storage Controller Device-Unit Information
    (DEV) Device Name [1-16 Characters]
--->(NAM) Device-Unit Name [1-14 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]      DOUBLE
    (SDS) Single or Double Sided Disks [Single/Double]        DOUBLE
    (EFI) 8 or 5 inch Disks [8/5]                             8
    (SUF) Standard or Uniform Format [Standard/Uniform]       STANDARD
    (GRA) Granularity [0-0FFFFH]                              0100H
    (DSZ) Device Size [0-0FFFFFFFFH]                          07C500H
--->(UN)  Unit Number on this Device [0-0FFH]                 0000H
    (UIN) Unit Info Name [1-16 Chars]
    (RUT) Request Update Timeout [0-0FFFFH]                   0096H
    (NB)  No. of Buffers [nonrand = 0/rand = 1-0FFFFH]        0008H
    (CUP) Common Update [True/False]                          TRUE
    (MB)  Max Buffers [0-0FFH]                                0FFH
```

Figure A-3. Device-Unit Information Screen (NAM and UN)

A.4 HOW TO EXCLUDE AUTOMATIC BOOT DEVICE RECOGNITION

To configure a system that does not include the ABDR feature, set the ABR prompt in the "EIOS" screen to "no" (see Figure A-1). This disables the ABDR feature.

When you set ABR to "no", the ICU will not display the ABDR screen. Therefore, you must provide information for the DLN, DPN, DFD, and DO prompts as input to the "Logical Names" screen. Figure A-4 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-4 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 device controller.

```
    (LOGN)      Logical Names
    Logical Name = log_name,device_name,file_driver,owners-id
            [1-12 Chars], [1-14 Chars], [P/S/N/R],[0-0FFFFH]
    [1] Logical Name = BB        , BB     , PHYSICAL,      0H
    [2] Logical Name = STREAM    , STREAM, STREAM   ,      0H
    [3] Logical Name = LP        , LP     , PHYSICAL,      0H
--->[4] Logical Name = SD        , AF0    , NAMED    ,      0H
```

Figure A-4. Logical Names Screen

APPENDIX B
PROMMING THE BOOTSTRAP LOADER AND THE iSDM™ MONITOR

B.1 INTRODUCTION

Chapter 2 stated that one of the ways to prepare the Bootstrap Loader for use is to combine it with one of the Intel monitor packages and burn the combined code into PROM. This appendix supplies information about combining the Bootstrap Loader and the iSDM monitor. The *iSDM System Debug Monitor User's Guide* also contains information about this process.

B.2 INCORPORATING THE iSDM MONITOR

This section gives the instructions required to place the first stage and the iSDM monitor into two 27128 EPROM devices. You can modify this example to suit your own purposes, or you can follow it exactly. Refer to the iPPS PROM PROGRAMMING SOFTWARE USER'S GUIDE for detailed information about the commands. The step-by-step procedure is as follows:

1. Enter the name of the (version 1.4 or later) software used with the iUPP Universal PROM Programmer:

2. Specify that the PROMs are 27128 EPROM devices:

3. Respond with the number of the desired work file drive:

   1

   This says that drive :f1: will be used for creating temporary IPPS workfiles.

4. Initialize the file type to be loaded:

   This says that the load file is an 8086 Object Module Format file (which the first stage and the iSDM monitor are).

5. 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 FE400H is an example value which is compatible with most configurations of the iSDM R3.2 monitor. The upper bound of the format range is 0FFF7FH, the highest memory location the Bootstrap Loader can use when combining it with the iSDM monitor. The upper bound also applies to all previous versions of the iSDM 86 or iSDM 286 monitors. Always check the monitor and Bootstrap Loader memory maps before burning the addresses into the PROM devices. Also, be sure that the addresses you use do not collide. The numbers 3, 2, and 1 match IPPS prompts for defining the information.)

6. Tell the software to program one EPROM with even-addressed bytes. Use the following formula to determine the address to use:

   address = ((address of first stage) - (start address of EPROM pair))/2

   Therefore:

   address = (FE400H - F8000H)/2 = 3200H

   The IPPS command is as follows:

7. Do the same for the odd-numbered bytes.

8. Exit the IPPS program.

As a further example for step number six above, the formula below determines the address to specify when using 27512 EPROM devices:

address = (FE400H - 0E0000H)/2 = 0F2000H

INDEX

%Again 3-27, 29
%Auto 3-14
%Auto_configure_memory 3-10
%B208 3-32
%B215 3-32
%B218A 3-33
%B220 3-32
%B251 3-35
%B254 3-36
%B264 3-36
%BIST 3-8
%BMPS 3-11, 4-5
%BSCSI 3-37
%BSERR.A86 3-27
%CICO 3-18
%Clear_SDM_extensions 3-18
%Console 3-14, 27, 28, 7-1, 2
%CPU 3-11, 4-5, 10
%Defaultfile 3-17
%Device 3-24, 4-6, 5-34, 6-8
%End 3-27, 31, 4-5, 12
%Halt 3-27, 30, 4-5, 10
%iAPX_186_INIT 3-13
%Installation 4-5, 11
%INT1 3-27, 29, 4-5, 9
%INT3 3-27, 30, 4-5, 10
%List 3-27, 28, 7-1, 2
%Loadfile 3-16
%Manual 3-14
%Retries 3-17
%SASI_unit_info 3-39, 4-7
%Serial_channel 3-19
%Text 3-27, 28, 7-1, 2

A

Actions taken by the Bootstrap Loader after an error 7-1
Automatic boot device recognition A-1, 2, 5
Automatically configuring memory 3-10

B

B208.A86 3-2, 31
B215.A86 3-2, 31, 4-2
B218A.A86 3-2, 31
B251.A86 3-2, 31
B254.A86 3-2, 31
B264.A86 3-2, 31, 4-2
BG3.A86 4-2
  Default file 4-5
  Editing 4-3
  Excluding macros 4-3
BG3.CSD 4-2
  Default file 4-14
  Invocation 4-15
  Modification 4-14
Board-scan algorithm 5-32
Boot device recognition A-1
Booting iRMX® I and iRMX® II Operating Systems from the same volume 1-4
Bootstrap Loader communication system 5-8, 6-8
Bootstrap Loader driver COMPACT sub-system 5-31
Bootstrap Loader failures 7-1, 6
BS$BROADCAST 5-20, 21
BSGETINTERCONNECT 5-22, 23
BS$RECEIVE 5-18
BS$SEND 5-15, 16
BS$SEND$RSVP 5-11, 12
BSSETINTERCONNECT 5-22, 26, 27
BS1.A86 3-1, 2, 5-33
  Custom drivers 5-33
  Editing 3-7
BS1.CSD 3-2, 41
  Default file 3-42
  Invocation 3-46
  Modification 3-44
BS3.A86 4-2, 6-8
  Editing 4-3
  Excluding macros 4-3
  Modification 6-8

BS3.CSD 4-2, 13
Default file 4-13
Invocation 4-15
Modification 4-14
BSCSI.A86 3-2, 31
BSERR.A86 3-1, 7-1
Built-In Self Test (BIST) 3-8

C

Chip mode configuration 3-13
Choosing a third stage 2-6
CI routines 3-18, 19
Clearing iSDM monitor command extensions 3-18
CO routines 3-18, 19
Condition codes
BS$BROADCAST 5-22
BSGETINTERCONNECT 5-26
BS$RECEIVE 5-20
BS$SEND 5-18
BS$SEND$RSVP 5-15
BSSETINTERCONNECT 5-30
Configuration 3-1, 4-1
CPU board 4-10
Files 3-31
Files for custom drivers 5-5
First stage 3-1
Memory 3-10
Message passing system 3-11, 4-5
Processor board type 4-10
Third stage 4-1
Controlling error message display 3-28
Conventions iv
CPU board configuration 4-10
CPU type 3-11, 4-10
CS register integrity 6-7

Custom drivers 5-1, 6-7
BS1.A86 alterations 5-33
BS3.A86 alterations 6-8
Configuration files 5-5
Determining processing mode 6-2
Device initialize 5-2, 6-3
Device read 5-3, 6-4
First stage configuration 5-4
First stage considerations 5-30
First stage requirements 5-1
Generating the first stage 5-34
Generating the third stage 6-9
Hard-coded configuration 5-4
MULTIBUS® II transport protocol 5-8, 6-8
Protected mode 6-6
Third stage configuration 6-7
Third stage requirements 6-2

D

Debug option 2-3
Default BG3.A86 file 4-5
Default BG3.csd file 4-14
Default BS1.CSD 3-42
Default BS3.CSD file 4-13
Default BSERR.A86 file 3-27
Default load file 3-17
Defining bootable devices
First stage 3-24
Third stage 4-6
Defining SASI bus initialization sequences 3-39, 4-8
Device driver 1-2, 7, 8
Code considerations 5-30
Configuration files 3-31
First stage 3-25
Device initialization
Requirements, first stage 5-2
Requirements, third stage 6-3
Procedure 5-1, 2, 6-1, 3
Device read
Procedure 5-1, 3, 6-1, 4
Requirements, first stage 5-3
Requirements, third stage 6-5
Device-specific third stage 1-5

Displayed error messages 7-2
Displaying error messages 3-28
Displaying the load file pathname 3-16
Drivers, custom 5-1

E

Editing BS1.A86 3-7, 5-33
Errors
  Analyzing 7-2, 5
  Bootloading 3-27
  Code 01 7-2
  Code 11 7-3
  Code 12 7-3
  Code 13 7-3
  Code 21 7-3
  Code 22 7-4
  Code 23 7-4
  Code 24 7-4
  Code 31 7-4
  Code 32 7-4
  Code 33 7-4
  Code 34 7-5
  Code 35 7-5
  Controlling message display 3-28
  Handling 7-1
  Initialization 7-7
  Message display 3-28
  Procedures 3-1, 8, 27, 7-1
ES register integrity 6-7
Examples
  Board-scan algorithm 5-32
  BS$BROADCAST 5-21
  BSGETINTERCONNECT 5-24
  BS$RECEIVE 5-19
  BS$SEND 5-16
  BS$SEND$RSVP 5-13
  BSSETINTERCONNECT 5-28
  Maintaining DS register integrity 6-6
  Modified BS1.A86 file 5-34
Excluding a device driver 3-26, 45
Excluding automatic boot device recognition A-5
Excluding BS1.A86 macros 3-7

F

Failures 7-1
First stage 1-1, 2
  BS1.CSD file 3-42
  BSERR.A86 file 3-27
  Configuration 3-1
  Custom drivers 5-1, 30
  Defining a bootable device 3-24
  Device driver configuration files 3-2
  Device drivers 1-7
  Device initialization 5-1, 2
  Device read 5-1, 3
  Error procedures 7-1
  Failure 7-6
  Generation 3-41
  Generation for custom drivers 5-34
  Halting the boot 3-30
  Initialization requirements 5-2
  iSDM™ monitor inclusion B-1
  Location 1-2, 8, 9, 3-47, 4-16
  Placing in memory 2-4
  Read requirements 5-3
  Second stage identification 3-14
  Size 1-2
  Steps when supplying your own drivers 3-40
  Supported device drivers 3-25
  User-supplied drivers 3-40

G

Generation
  First stage containing a custom device driver 5-34
  Third stage 4-13
  Third stage (custom device driver) 6-9
Generic third stage 1-4

H

Halting the Bootstrap Loader during errors
  First stage 3-30
  Third stage 4-10
Hard-coding custom driver configuration information 5-4
How to configure the first stage 3-1
How to configure the third stage 4-1
How to define a device to boot from, first stage 3-24

How to display the load file pathname 3-16
How to exclude automatic boot device recognition A-5
How to include automatic boot device recognition A-2
How to indicate a default load file 3-17

I

Identifying the serial channel 3-19
Including automatic boot device recognition A-2
Incorporating the iSDM™ monitor B-1
Initialization errors 7-7
Intel-supplied BG3.A86 file 4-5
Intel-supplied Bootstrap Loader device drivers 1-8
Intel-supplied device drivers 3-31
Intel-supplied first stage drivers 3-25
Intel-supplied third stage drivers 4-7
Intel-supplied third stage files 2-7
Interconnect space 5-22
Interrupt
    INT1 3-29, 4-9
    INT3 3-30, 4-10
Invocation from the iSDM monitor 2-2
Invoking the BG3.CSD submit files 4-15
Invoking the BS1.CSD submit file 3-46
Invoking the BS3.CSD submit files 4-15
iSBC® 208 Driver 3-25
iSBC® 215 Driver 3-25, 4-7
iSBC® 251 Driver 3-25
iSBC® 254 Driver 3-25
iSBC® 264 Driver 3-25, 4-7
iSBX® 218A Driver 3-25
iSDM™ monitor B-1
iSDM™ monitor command extensions, clearing 3-18

L

Load file 1-5
    Device 3-14
    Pathname specification 2-1
Loading the Bootstrap Loader into memory 2-4
Location of first stage 1-2
Location of second stage 1-3

M

Manual overview iii
Memory locations used by the Bootstrap Loader 1-8, 3-47, 4-16
Message broadcasting 5-20
Message passing system configuration 3-11, 4-5
Message types 5-10
Modifying the BG3.CSD submit files 4-14
Modifying the BS1.CSD submit file 3-44
Modifying the BS3.CSD submit files 4-14
Monitor entry after third stage 4-11
MULTIBUS® II environment 5-30, 6-2
MULTIBUS® II transport protocol 5-1, 8, 6-2, 8

N

Naming the load file 1-5
Naming the third stage 1-5

O

Operator's role 2-1

P

Placing the Bootstrap Loader into memory 2-4
Product overview iii, 1-1
Programmatically loading the first stage 2-5
Promming the Bootstrap Loader and the iSDM™ monitor B-1
Protected mode considerations 6-6

R

Reader level iii
Receive transaction model 5-15
Request/response transaction model 5-10
Retries for ready devices 3-17

S

SASI bus initialization sequence definition 3-39, 4-8
SASI controller 3-39, 4-7
SCSI controller 3-39, 4-7
SCSI driver 3-25
Searching for a ready device 3-17

Second stage 1-1, 3
Error procedures 7-1
Failure 7-6
Location 1-3, 8, 9, 3-47, 4-16
Size 1-3
Send transaction model 5-15
Serial channel identification 3-19
Serial communication
Base port 3-20
Baud counter 3-21
Counter base port 3-21
Counter type 3-21
Flags 3-22
Serial communication 3-20
Error messages 3-23
Serial controller device 3-20
Software interrupt (INT1) 3-29, 4-9
Software interrupt (INT3) 3-30, 4-10
SS register integrity 6-7
Stages 1-2
Supplying configuration information
First stage driver 5-4
Third stage driver 6-7
Supplying your own device driver 3-40, 4-12
Supported 5.25-inch diskettes 3-26
Supported 8-inch diskettes 3-26
Supported device drivers 3-25, 31
Supported devices 1-8

T

Third stage 1-2, 4
BG3.CSD file 4-14
BS3.CSD file 4-13
Choosing 2-6
Configuration files 4-2
Custom drivers 6-1, 6-8
Defining a bootable device 4-6
Device drivers 1-7
Device initialization 6-3
Device read 6-4
Device-specific 1-5, 4-1
Error procedures 7-1
Failure 7-6
Generation 4-13

Third stage (cont.)
Generation for custom drivers 6-9
Generic 1-4, 4-1
Halting the boot 4-10
Initialization requirements 6-3
Intel-supplied 2-7
Location 1-4, 8, 9, 4-16
Naming 1-5
Read requirements 6-5
Steps for supplying your won drivers 4-12
User-supplied drivers 4-12
Transaction ID value 5-22
Transmission
Modes 5-22
Status 5-22

U

User-supplied drivers 3-40, 4-12
Using the Bootstrap Loader 2-1
Using the iSDM debug option 2-4

W

Writing a custom first stage driver 5-1
Writing a custom third stage driver 6-1

intel

EXTENDED iRMX® II SYSTEM DEBUGGER REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

INTRODUCTION

The iRMX II System Debugger is a memory-resident extension of the iSDM™ System Debug Monitor and the D-MON386 Monitor. The System Debugger gives you a static debugging tool that can recognize and display all iRMX II objects. It enables you to examine your iRMX II system interactively so you can find and correct errors.

READER LEVEL

This manual is intended for application engineers familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide* and system programmers implementing device drivers, object managers, and operating system extensions.

MANUAL OVERVIEW

This manual consists of the following chapters:

Chapter 1	INTRODUCTION--This chapter describes the features of the System Debugger, illustrates how the System Debugger relates to EPROM-based debugging tools, and explains how to use the System Debugger. Read this chapter if you are going through the manual for the first time.
Chapter 2	SYSTEM DEBUGGER COMMANDS--This chapter contains detailed descriptions of the System Debugger commands, presented in alphabetical order. When debugging your system, refer to this chapter for specific information about the format and parameters of the commands.

Chapter 3	SAMPLE DEBUG SESSION--This chapter shows in a step-by-step fashion how to use System Debugger features. The chapter contains a sample debugging session demonstrating how to use iSDM monitor and System Debugger commands to locate an application-code error, correct it, and test the change. Discrete examples showing additional debugging techniques are also included. Use this chapter as a hands-on introduction to the System Debugger.
Appendix A	iSDM MONITOR COMMANDS--This appendix briefly describes the function of all basic iSDM monitor commands. Use this appendix as a quick reference to the iSDM monitor. For more information see the *iSDM System Debug Monitor User's Guide*.
Appendix B	D-MON386 MONITOR COMMANDS--This appendix briefly describes the function of all basic D-MON386 monitor commands. For more information, refer to the *D-MON386 Debug Monitor for the 80386 User's Guide*.

CONVENTIONS

This manual uses the following format conventions:

- User input appears in one of the following forms:

```
as bolded text within a screen
```

- The text <CR> appears where you must enter a carriage return. When pressing the carriage return key, the text <CR> does not appear on the console.
- Although all syntax diagrams show uppercase letters (e.g., VR), you can also use lowercase letters.
- The manual refers to the iRMX II Operating System as the operating system.
- All numbers unless otherwise stated are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).
- Darker shaded text appearing over shaded text within figures or screen displays does not actually appear on the screen. The text within the darker box supplies information that is helpful in understanding the figure or screen display.

Intel®

CONTENTS

CHAPTER 1
iRMX® II SYSTEM DEBUGGER OVERVIEW **PAGE**

1.1 Introducing the iRMX® II System Debugger 1-1
1.2 Supporting the System Debugger 1-2
1.3 Configuring the System Debugger 1-2
1.4 Invoking the System Debugger 1-2
1.5 Using the System Debugger 1-3
1.6 Returning to Your Application 1-4

CHAPTER 2
SYSTEM DEBUGGER COMMANDS **PAGE**

2.1 Introduction 2-1
2.2 Checking Validity of Tokens 2-1
2.3 Pictorial Representation of Syntax 2-2
2.4 Leaving the Monitor 2-3
2.5 Command Directory 2-4
VB--Display DUIB Information 2-5
VC--Display System Call Information 2-9
VD--Display a Job's Object Directory 2-12
VF--Display Number of Free Slots 2-14
VH--Display Help Information 2-16
VJ--Display Job Hierarchy 2-18
VK--Display Ready and Sleeping Tasks 2-22
VO--Display Objects in a Job 2-24
VR--Display I/O Request/Result Segment 2-27
VS--Display Stack and System Call Information 2-31
VT--Display iRMX® II Object 2-36
Job Display 2-37
Task Display 2-39
Mailbox Display 2-42
Semaphore Display 2-44
Region Display 2-45
Segment Display 2-46
Extension Object Display 2-47
Composite Object Display 2-47
Buffer Pool Display 2-60
VU--Display System Calls in a Task's Stack 2-62

CHAPTER 3 **PAGE**
SAMPLE DEBUG SESSION

3.1 Introduction 3-1
3.2 Sample Program 3-1
3.3 Debugging the Program 3-9
3.4 Viewing System Objects 3-17

APPENDIX A **PAGE**
ISDM™ MONITOR COMMANDS

A.1 Introduction A-1
A.2 Command Directory A-1
A.3 Command Descriptions A-3
A.3.1 B--Bootstrap Load A-3
A.3.2 C--Compare A-4
A.3.3 D--Display Memory/Descriptor Tables/Disassembled Instructions A-4
A.3.4 E--Exit A-4
A.3.5 F--Find A-5
A.3.6 G--Go A-5
A.3.7 I--Port Input A-5
A.3.8 K--Echo File A-5
A.3.9 L--Load Absolute Object File A-6
A.3.10 M--Move A-6
A.3.11 N--Execute Single Instructions A-6
A.3.12 O--Port Output A-7
A.3.13 P--Print A-7
A.3.14 Q--Enable Protection (80286/80386 only) A-7
A.3.15 R--Load and Go A-8
A.3.16 S--Substitute Memory/Descriptor Table Entry A-8
A.3.17 X--Examine/Modify Registers A-9
A.3.18 Y--Symbols (80286 or 80386 only) A-9

APPENDIX B **PAGE**
D-MON386 COMMANDS

B.1 Introduction B-1
B.2 Entering Commands B-1
B.3 Command Directory B-2
B.4 Command Descriptions B-4
B.4.1 $ B-5
B.4.2 ASM B-5
B.4.3 B B-5
B.4.4 BASE B-5
B.4.5 BYTE B-5
B.4.6 COUNT/ENDCOUNT B-5
B.4.7 CREGS B-6

APPENDIX B (continued) **PAGE**

B.4.8 DPORT....B-6
B.4.9 DT....B-6
B.4.10 DWORD....B-6
B.4.11 EVAL....B-6
B.4.12 FLAGS....B-6
B.4.13 GDT....B-7
B.4.14 GO....B-7
B.4.15 HELP....B-7
B.4.16 HOST....B-7
B.4.17 IDT....B-7
B.4.18 INTn....B-7
B.4.18 ISTEP....B-8
B.4.19 LDT....B-8
B.4.20 N0-N9....B-8
B.4.21 ORDn....B-8
B.4.22 PD....B-8
B.4.23 PORT....B-9
B.4.24 Register-name....B-9
B.4.25 REGS....B-9
B.4.26 SREGS....B-9
B.4.27 SWBREAK....B-9
B.4.28 SWREMOVE....B-10
B.4.29 TSS....B-10
B.4.30 USE....B-10
B.4.31 VERSION....B-10
B.4.32 WORD....B-10
B.4.33 WPORT....B-11

Intel®

FIGURES

FIGURE **PAGE**

2-1 Format of VB Output....2-5
2-2 Format of VC Output....2-9
2-3 Format of VD Output....2-12
2-4 Format of VF Output....2-14
2-5 Format of VH Output....2-17

FIGURE PAGE

2-6 Format of VJ Output 2-18
2-7 iRMX® II Job Tree 2-20
2-8 Format of VK Output 2-22
2-9 Format of VO Output 2-24
2-10 Format of VR Output 2-28
2-11 Format of VS Output 2-32
2-12 Format of VT Output: Job Display 2-37
2-13 Format of VT Output: Non-Interrupt Task 2-39
2-14 Format of VT Output: Interrupt Task 2-39
2-15 Format of VT Output: Mailbox with No Queue 2-42
2-16 Format of VT Output: Mailbox with Task Queue 2-42
2-17 Format of VT Output: Mailbox with Object Queue 2-43
2-18 Format of VT Output: Mailbox with Data Message Queue 2-43
2-19 Format of VT Output: Semaphore with No Queue 2-44
2-20 Format of VT Output: Semaphore with Task Queue 2-45
2-21 Format of VT Output: Region with No Queue 2-45
2-22 Format of VT Output: Region with Task Queue 2-46
2-24 Format of VT Output: Extension Object 2-47
2-23 Format of VT Output: Segment 2-46
2-25 Format of VT Output: Composite Object Other Than BIOS 2-48
2-26 Format of VT Output: BIOS User Object Composite 2-49
2-27 Format of VT Output: BIOS Physical File Connection 2-49
2-28 Format of VT Output: BIOS Stream File Connection 2-53
2-29 Format of VT Output: BIOS Named File Connection 2-54
2-30 Format of VT Output: BIOS Remote File Connection 2-56
2-31 Format of VT Output: Signal Protocol Port 2-57
2-32 Format of VT Output: Data Transport Protocol Port 2-58
2-33 Format of VT Output: Data Transport Protocol Port 2-58
2-34 Format of VT Output: Buffer Pool 2-60
2-35 Format of VU Output 2-63
3-1 Example PL/M-286 Application (Init) 3-2
3-2 Example PL/M-286 Application (Alphonse) 3-5
3-3 Example PL/M-286 Application (Gaston) 3-7
3-4 MOVW in Gaston Code 3-12

CHAPTER 1
iRMX® II SYSTEM DEBUGGER OVERVIEW

1.1 INTRODUCING THE iRMX® II SYSTEM DEBUGGER

When you develop application systems, you need debugging capabilities on your development system. In addition to the iSDM System Debug Monitor or the D-MON386 Monitor, Intel provides the iRMX II System Debugger (SDB) for debugging your iRMX II-based application system.

NOTE

The remainder of this manual uses the term "monitor" to refer to both the iSDM System Debug Monitor and the D-MON386 Monitor.

The System Debugger is a memory-resident extension of the monitor; therefore, you must have the monitor if you have the System Debugger configured into your system. The monitor provides code disassembly, execution breakpoints, memory display, and program download capabilities. The System Debugger extends the monitor's disassembly functions by interpreting iRMX II calls, data structures, and stacks.

Monitor and System Debugger commands are entered in response to the iSDM Monitor's protected-mode prompt (..) or the D-MON386 Monitor's prompt (>). When you invoke the monitor, both the operating system and your application system are frozen. As you use monitor commands to set breakpoints while the application code is executed, you can inspect system objects, change system call parameters and registers, and test changes. Refer to Appendix A for more information on iSDM Monitor commands and Appendix B for D-MON386 Monitor commands.

1.2 SUPPORTING THE SYSTEM DEBUGGER

To use the System Debugger, you must have one of the following hardware configurations with all the required support hardware:

- An Intel Microcomputer connected to an 80286- or 386-based board
- A terminal connected directly to an 80286- or 386-based board
- An Intellec® Development System connected to an 80286- or 386-based board

In addition to the above hardware, you must have both of the following:

- The EPROM portion of the iSDM System Debug Monitor or the D-MON386 Monitor
- At least the minimal configuration of the iRMX II Nucleus

1.3 CONFIGURING THE SYSTEM DEBUGGER

You cannot use the System Debugger until you include it in your system through the Interactive Configuration Utility (ICU). To include the System Debugger, begin by invoking the ICU. Next, provide the following information the ICU requires to configure the System Debugger:

1. In the ICU's "Sub-Systems" screen, respond "yes" to the SDB prompt.
2. In the ICU's "System Debugger" screen, set the interrupt level you want to use to invoke the monitor manually (by pressing a hardware interrupt button).

   To use the Non-Maskable Interrupt (NMI) for debugging device drivers, see the *Extended iRMX II Hardware and Software Installation Guide*.

For detailed information on configuring the System Debugger, consult the *Extended iRMX II Interactive Configuration Utility Reference Manual*.

1.4 INVOKING THE SYSTEM DEBUGGER

You must enter the monitor to use the System Debugger. You can invoke the monitor in three ways:

1. Use a hardware switch physically connected to the interrupt level you specified during configuration. Activating this switch halts the application system, saves the system's contents, and passes control to the monitor.
2. Use the Human Interface DEBUG command. DEBUG loads your specified application program into main memory and transfers control to the monitor.

3. Use the Bootstrap Loader DEBUG switch. When you specify this switch, the monitor comes up after the system is loaded but before the system starts running. The CS:IP points to the first instruction of the application system. At this point the system has not been initialized; therefore, you can run only monitor commands. Using the MAP286 output, you can identify where you want to insert breakpoints. (For more information on BIND, MAP, and OVL, see the *iAPX 286 Utilities User's Guide for iRMX II Systems*). Use the break address parameter in the monitor's GO (G) command to set breakpoints in the application system code. When you enter "G <CR>", the system starts and is initialized. The monitor is invoked when the CS:IP reaches the breakpoints. For more information on booting with DEBUG, consult the *Extended iRMX II Bootstrap Loader Reference Manual*.

When you invoke the monitor, the application system stops running and all system activity freezes. The appropriate prompt appears (the ".." for the iSDM Monitor or the ">" for the D-MON386 Monitor), and you can begin entering System Debugger and monitor commands to examine system objects.

1.5 USING THE SYSTEM DEBUGGER

The System Debugger uses monitor procedures to parse the command line and to output to the console; therefore, you run both System Debugger and monitor commands from the monitor. The syntax for System Debugger commands is a "V" or "v" followed by another letter, an optional space, and an optional parameter.

The twelve System Debugger commands (described in Chapter 2) fall into three categories:

- Eight commands extend the monitor memory display functions by displaying iRMX II data structures and objects.
- Three commands extend the monitor disassembly functions by recognizing and displaying iRMX II calls.
- A help command provides a short description of all the commands.

All commands either display information as hexadecimal numbers or try to interpret the information. If the System Debugger cannot interpret the information, it displays the actual hexadecimal value, followed by two question marks.

iRMX II provides two features that enable you to leave the monitor without resetting your system: warm-start and CLI-restart. The warm-start feature reinitializes the system and returns control to the Human Interface. The CLI-restart feature deletes the current job then returns control to the Command Line Interpreter. Refer to Chapter 2 for more information on these features.

1.6 RETURNING TO YOUR APPLICATION

When you have finished debugging your application system with the System Debugger, or if you want to test the changes you made to the application code, use the monitor's GO command (G) to resume execution of the application.

CHAPTER 2
SYSTEM DEBUGGER COMMANDS

2.1 INTRODUCTION

This chapter contains detailed descriptions of the iRMX II System Debugger commands. Commands appear in alphabetical order, with the first occurrence of each command appearing in at the top of the page. A directory of the commands, divided into functional groups, precedes the command descriptions.

This chapter uses the following conventions:

- "CS:IP" is the Code Segment:Instruction Pointer--The pointer to the instruction that would be executed next if the application system were running. If you specify an IP value (one four-digit hexadecimal number) but not a CS value, the System Debugger uses the current CS as the default base.
- "SS:SP" is the Stack Segment:Stack Pointer--The pointer to the current stack location.
- Entering zero (0) as a value for an optional parameter is the same as omitting the parameter; the default value of the parameter is used.
- All terminal examples assume that the iSDM System Debug Monitor is being used. Thus, example input lines show the iSDM monitor prompt (..).

2.2 CHECKING VALIDITY OF TOKENS

Many System Debugger commands use iRMX II tokens as parameters or display tokens as part of the command output. The iRMX II Operating System maintains tokens in doubly linked lists. When you enter a token as a parameter, the System Debugger checks the validity of the token by looking at the forward and backward links of the token.

If one of the links is bad, the System Debugger generates an error message along with the standard command output. The token you enter as a parameter always appears as the center value in each line of the token display. The displays for forward- and backward-link errors are as follows:

Forward link ERROR:	4111-->4E85	4111<--4E85-->4155	?FFFF<--4155
Backward link ERROR:	4111-->410F?	4111<--4E85-->4155	4E85<--4155

Arrows to the left indicate backward links; arrows to the right indicate forward links. A question mark before or after a value signifies a forward or backward link error, respectively.

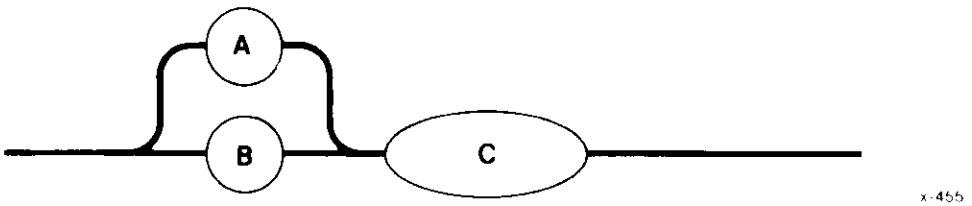
If both links are bad, the System Debugger considers the token invalid. A token may also be invalid if it belongs to an object in the deletion process, if an incorrect token is entered as a parameter in a system call, or if a deleted or unused token is entered as a parameter. When the token is invalid, the System Debugger displays the following message:

```
*** INVALID TOKEN ***
```

A link error indicates that iRMX II data structures have been corrupted. The most common reason for this problem is a task might have accidentally written over part of the system data structures. However, the iRMX II protection mode feature protects against such overwriting under normal circumstances. Data structure corruption can also occur if you are using the Non-Maskable Interrupt (NMI). The Nucleus may have been interrupted while it was setting up the links. (The NMI is a hardware interrupt. For more information on the NMI, see the *80286 Hardware Reference Manual* or the *80386 Hardware Reference Manual.*)

2.3 PICTORIAL REPRESENTATION OF SYNTAX

This chapter uses a schematic device to illustrate command syntax. The schematic consists of what looks like an aerial view of a model railroad, with syntactic elements (appearing in circles) scattered along the track. To construct a valid command, imagine that a train enters the system at the far left, travels from left to right only (backing up is not allowed), chooses one branch at each fork, and finally departs at the far right. The command generated consists of the syntactic elements it encounters on its journey. The following schematic shows two valid sequences: AC and BC.



These schematics do not show spaces as elements, but you may include one or more spaces between the command and parameter. For example, even though the syntax for VR is as follows:

x 456

The following command is valid:

```
. .
```

The space between "VR" and "xxxx" is optional.

2.4 LEAVING THE MONITOR

Two features enable you to leave the monitor without resetting your system: warm-start and CLI-restart.

The warm-start feature is the process of starting a system without reloading it from secondary storage. Warm-start reinitializes the system, that is, it begins executing the application system at the same point where the Bootstrap Loader passes control to the system.

To warm-start the system from the iSDM monitor, enter the following command:

```
. .
```

To warm-start the system from the D-MON386 monitor, enter the following command:

```
>
```

If no system code or data segments were corrupted, the system reinitializes. If segment corruption has occurred, the application system will not run; you must reboot the system.

If your system contains a Command Line Interpreter, and running your application program causes an exception that breaks to the monitor (for example, a General Protection exception), enter the following command to CLI-restart the system from the iSDM monitor:

```
. .
```

Enter the following command to CLI-restart the system from the D-MON386 monitor:

```
>
```

These commands causes the system to attempt to delete the job tree of the running task. If the running task is part of the application's job (not a subsystem task running on behalf of the job) control returns to the Command Line Interpreter. Otherwise, you must reboot the system.

2.5 COMMAND DIRECTORY

Command — Page

DISPLAYING iRMX II DATA STRUCTURES

VB--Display DUIB Information.....2-5

VD--Display a Job's Object Directory.....2-12

VF--Display Number of Free Slots.....2-14

VJ--Display Job Hierarchy.....2-18

VK--Display Ready and Sleeping Tasks.....2-22

VO--Display Objects in a Job.....2-24

VR--Display I/O Request/Result Segment.....2-27

VT--Display iRMX II Object.....2-36

RECOGNIZING AND DISPLAYING iRMX II SYSTEM CALLS

VC--Display System Call Information.....2-9

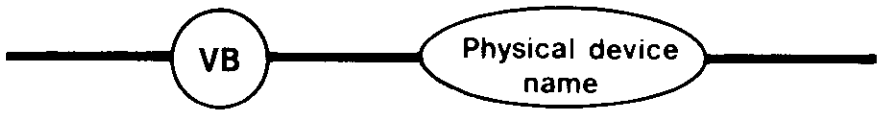
VS--Display Stack and System Call Information.....2-31

VU--Display System Calls in a Task's Stack.....2-62

OTHER COMMANDS

VH--Display Help Information.....2-16

The VB command displays the DUIB information for the specified physical device. For additional information about Device-Unit Information Blocks (DUIBs), refer to Chapter 4 of the *Extended iRMX II Device Drivers User's Guide*.



PARAMETER

Physical device	The name of the physical device for which you want to view the DUIB information (e.g., WMF0). This device must be part of the system configuration.

DESCRIPTION

The VB command displays the DUIB information for the specified physical device. Figure 2-1 illustrates the output from the VB command.

```
Device name:        <physical device name>

Functs:             xx                  DUIB address    xxxx:xxxx
Dev$gran            xxxx                Max$buffers     xx
Dev$size            xxxxxxxx            Device          xx
Unit                xx                  Dev$unit        xxxx
Device$info$p       xxxx:xxxx           Unit$info$p     xxxx:xxxx
Update$timeout      xxxx                Num$buffers     xxxx
Priority            xx                  Fixed$update    xx
Init$io             xxxx:xxxx           Finish$io       xxxx:xxxx
Queue$io            xxxx:xxxx           Cancel$io       xxxx:xxxx

Flags:              xx                  Valid
  Density           xxxxxx              Sides           xxxxxx
  Size              x                   Format          xxxxxxxx

File driver:        xxxx                Named           xxxx
  Physical          xxxx                Stream          xxxxx
```

Figure 2-1. Format of VB Output.

The fields displayed in Figure 2-1 are as follows:

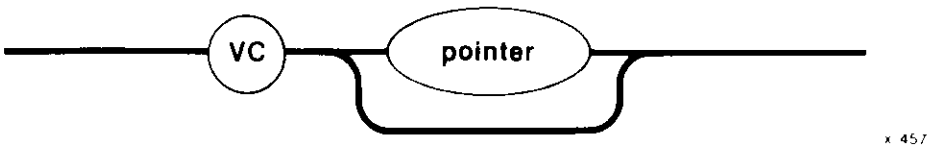
Functs	A BYTE used to specify the I/O function validity for this device-unit.
DUIB address	The starting address in memory of the specified DUIB.
Dev$gran	A WORD that specifies the device granularity, in bytes. This parameter applies to random access devices, and to some common devices, such as tape drives. It specifies the minimum number of bytes of information that the device reads or writes in one operation.
Max$buffers	The maximum number of buffers that the EIOS can allocate for a connection to this device-unit when the connection is opened by a call to S$OPEN.
Dev$size	The number of bytes of information that the device-unit can store.
Device	The number of the device with which this device-unit is associated.
Unit	The number of this device-unit, which distinguishes this unit from other units of the device.
Dev$unit	The device-unit number, which distinguishes this device-unit from other device-units in the hardware system.
Device$info$p	A POINTER to a structure that contains additional information about the device. The common, random, and terminal device drivers require a Device Information Table in a specific format, for each device.
Unit$info$p	A POINTER to a structure that contains additional information about the unit. Random access, common device (such as tape drives), and terminal device drivers require this Unit Information Table in a specific format.
Update$timeout	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.
Num$buffers	The number of buffers of device-granularity size that the I/O System allocates.
Priority	The priority of the I/O System service task for the device.
Fixed$update	Indicates whether the fixed update option was selected for this device-unit when the application system was configured.
Init$io	The address of the Initialize I/O procedure associated with this unit.

Finish$io	The address of the Finish I/O procedure associated with this unit.
Queue$io	The address of the Queue I/O procedure associated with this unit.
Cancel$io	The address of the Cancel I/O procedure associated with this unit.
Flags	Specifies the characteristics of diskette devices.
Valid	Indicates whether the Flags field is "Valid" or "Not Valid" for this device.
Density	The density of the device. If the flags for this DUIB are invalid, this field is marked "N/A".
Sides	The number of media sides that the device can write to. If the flags for this DUIB are invalid, this field is marked "N/A".
Size	The physical size of the device (5 1/4-inch or 8-inch). If the flags for this DUIB are invalid, this field is marked "N/A".
Format	Indicates whether track 0 of a disk is to be formatted as a STANDARD diskette (128 bytes/sector) or as a UNIFORM diskette (all sectors formatted as specified). This parameter applies only to flexible diskettes. Hard disks are always specified as UNIFORM. If the flags for this DUIB are invalid, this field is marked "N/A".
File driver:	A WORD that indicates the BIOS file driver to which this connection is attached.
Named	Indicates whether this device is configured to use the Named file driver.
Physical	Indicates whether this device is configured to use the Physical file driver.
Stream	Indicates whether this device is configured to use the Stream file driver.

ERROR MESSAGES

Syntax Error	An error was made when entering the command. The correct syntax is VB <physical device>. Any other syntax produces this message.
VB not supported	VB couldn't find the byte bucket DUIB entry in the BIOS code segment. If no DUIB entry for the byte bucket exists, VB is unsupported. If the BIOS has not been configured into the system, or if the BIOS code segment has execute-only attributes, this error message is returned.
DUIB not found	VB returns this error message under these conditions: 1. The DUIB is not configured into the system. 2. The DUIB entry for the specified device is located before the byte bucket DUIB entry. 3. The user made an error while entering the physical device name.

The VC command checks to see if a CALL instruction is an iRMX II system call. The VC command identifies system calls for all iRMX II Operating System layers.



PARAMETER

pointer — The address of the CALL instruction to be checked. This parameter can be any valid monitor address (two four-digit hexadecimal numbers separated by a colon).

If you are using the iSDM monitor and you do not supply a pointer (or you specify 0), this parameter defaults to the current CS:IP. If you specify an IP value (one four-digit hexadecimal number) but not a CS value, the System Debugger uses the current CS as the default base.

If you are using the D-MON386 monitor and you specify the address with an offset value with no base value, the parameter defaults to the current CS:IP value.

DESCRIPTION

If the CALL instruction is an iRMX II system call, the VC command displays information about the CALL instruction as shown in Figure 2-2.

```
gate #NNNN
(subsystem)system call
```

Figure 2-2. Format of VC Output

The fields in Figure 2-2 are as follows:

gate #NNNN — The gate number associated with the iRMX II system call at the address specified in the command.

(subsystem)	The iRMX II Operating System layer corresponding to the system call.
system call	The name of the iRMX II system call.

NOTE

The System Debugger uses the gate number to determine whether the CALL instruction represents a system call. Since the System Debugger does not disassemble the code, but rather examines a byte value at a particular offset from the CALL instruction, in rare cases a non-system call can be displayed as an iRMX II system call. However, the System Debugger does recognize and display all iRMX II system calls.

ERROR MESSAGES

Syntax Error	An error was made in entering the command.
Not a system CALL	The parameter specified points to a CALL instruction that is not an iRMX II system call.
Not a CALL instruction	The CS:IP specified does not point to any kind of call instruction.

EXAMPLES

Suppose you disassembled the following code using the iSDM monitor's Display Memory (DX) command:

```
18A0:006D 50         PUSH      AX
18A0:006E E8AD1E     CALL      A = 1F1E            ;$+7856
18A0:0071 E8DD03     CALL      A = 0451            ;$+992
18A0:0074 B80000     MOV       AX,0
18A0:0077 50         PUSH      AX
18A0:0078 8D060600   LEA       AX,WORD PRT 006
18A0:007C 1E         PUSH      DS
18A0:007D 50         PUSH      AX
18A0:007E E8411E     CALL      A = 1EC2            ;$+7748
18A0:0081 A30000     MOV       WORD PTR 0000H,AX
```

If you use the VC command on the CALL instruction at address 18A0:006E by entering the following command:

. .

The System Debugger displays the following information:

```
gate #0468
(Nucleus) set exception handler
```

Gate number 0468 corresponds to an RQSETEXCEPTION$HANDLER system call, which is a Nucleus call.

Now, suppose you want to see if the CALL instruction at 18A0:0071 is a system call. Enter the following command:

```
..
```

The System Debugger responds with the following:

```
Not a system CALL
```

Finally, if you use the VC command on the instruction at 18A0:0074, the System Debugger responds with the following:

```
Not a CALL instruction
```

The VD command displays a job's object directory.

x 458

PARAMETER

job token	The token for the job having the object directory you want displayed. To obtain the job token, use the VJ command.

DESCRIPTION

If you specified a valid job token, the System Debugger displays the job's object directory, as shown in Figure 2-3.

```
Directory size:  xxxx                Entries used:  xxxx

name1          token1
name2          tasks waiting         token2...tokeni
.              .
.              .
.              .
namej          tokenj
namek          tokenk
.              .
.              .
.              .
namen          tokenn
```

Figure 2-3. Format of VD Output

Figure 2-3 shows these fields:

Directory size	The maximum number of entries this job can have in its object directory.
Entries used	The number of entries presently in the directory.

name1...namen	The names under which objects are catalogued. These names were assigned at the time the objects were catalogued with RQ$CATALOG$OBJECT.
token1...tokenn	Tokens for the catalogued objects.
tasks waiting	Signifies that one or more tasks have performed an RQ$LOOKUP$OBJECT on an object not catalogued. The tokens following this field identify the tasks still waiting for the object to be catalogued.

For more information on object directories, see the *Extended iRMX II Nucleus User's Guide*.

ERROR MESSAGES

Syntax Error	No parameter was specified for the command, or an error was made in entering the command.
TOKEN is not a Job	A valid token was entered that is not a job token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).

EXAMPLE

Suppose you want to look at the object directory of job "2280". Enter the following command:

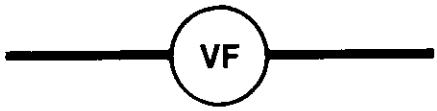
```
..
```

The System Debugger responds with

```
Directory size:  000A    Entries used:  0003

$              2228
R?IOUSER       2200
RQGLOBAL       2280
```

The symbols "$", "R?IOUSER", and "RQGLOBAL" are the names of objects the system creates; their respective tokens are 2228, 2200, and 2280. There are no waiting tasks or invalid entries.

The VF command displays the number of free Global Descriptor Table slots available to the user.



x-1863

PARAMETERS

The VF command has no parameters.

DESCRIPTION

The VF command displays the number of free Global Descriptor Table (GDT) slots available to the user, in the format shown in Figure 2-4.

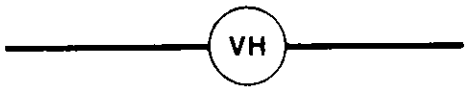
```
Number of free slots = xxxxxxxx
```

Figure 2-4. Format of VF Output.

ERROR MESSAGES

Syntax Error	An error was made in entering the command.

The VH command displays and briefly describes the twelve System Debugger commands.



x 459

PARAMETERS

This command has no parameters.

DESCRIPTION

The VH command lists all of the System Debugger commands, along with their parameters and descriptions.

ERROR MESSAGE

Syntax Error	An error was made in entering the command.

EXAMPLE

If you enter the following command:

. .

The System Debugger responds as shown in Figure 2-5.

```
Extended iRMX II SYSTEM DEBUGGER, Vx.y
Copyright <year> Intel Corporation

    vb <Dev Name>        Displays DUIB for physical device.
    vc [<POINTER>]       Display system call.
    vd <Job TOKEN>       Display job's object directory.
    vf                   Displays number of free slots available to user.
    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 II object.
    vu <task TOKEN>      Unwind task stack, displaying system calls.
```

Figure 2-5. Format of VH Output

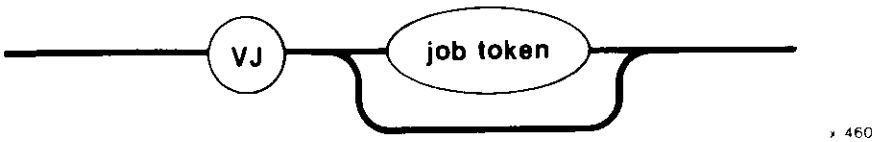
< >	Angle brackets surround required variable fields.
[< >]	Square and angle brackets surround optional fields.

NOTE

The system uses default values if you specify zero (0) for any of the optional parameters in Figure 2-5. Using zero for required parameters causes the system to display the following message:

```
Syntax Error
```

The VJ command displays the portion of the job hierarchy that descends from the level you specify.



PARAMETER

job token — The token of the job for which you want to display descendant jobs.

If you do not specify a job token, or you specify zero (0), VJ displays the root job and its descendant jobs.

If the job has more than 44 generations of job descendants, the System Debugger discontinues the display at the 44th descendant level, displays an error message, and prompts for another command.

DESCRIPTION

The VJ command displays the token of the specified job and the tokens of all its descendant jobs. It also displays the tokens of jobs (and their descendants) at the same level as the specified job. The tokens for descendant jobs are indented three spaces to show their job's position in the hierarchy. Figure 2-6 shows the format of the job hierarchy display.

```
iRMX® II Job Tree

token1                          Root Job
    token2                      Human Interface
        token3                  Command Line Interpreter
            token4              Application
    token5                      EIOS
    token6                      BIOS
```

Figure 2-6. Format of VJ Output

The fields in Figure 2-6 are

$token_1$	The token you specified as job token (recall that the root job token is the default).
$token_2$...$token_6$	The tokens for the descendant jobs of $token_1$.

In Figure 2-6, the Human Interface, EIOS, and BIOS Jobs are indented three spaces to signify that they are children of the Root Job. Similarly, the Command Line Interpreter Job is the child of the Human Interface Job, and the Application Job is the child of the Command Line Interpreter Job.

ERROR MESSAGES

Syntax Error	An error was made in entering the command.
TOKEN is not a Job	A valid token was entered that is not a job token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).
SDB job nest limit exceeded	The specified job (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 the following command:

```
..
```

Suppose the System Debugger responds with the following job tree:

```
            iRMX® II Job Tree

0258
    0F38
        1670
            2460
    0E88
    0E00
```

Figure 2-7 shows this job tree:

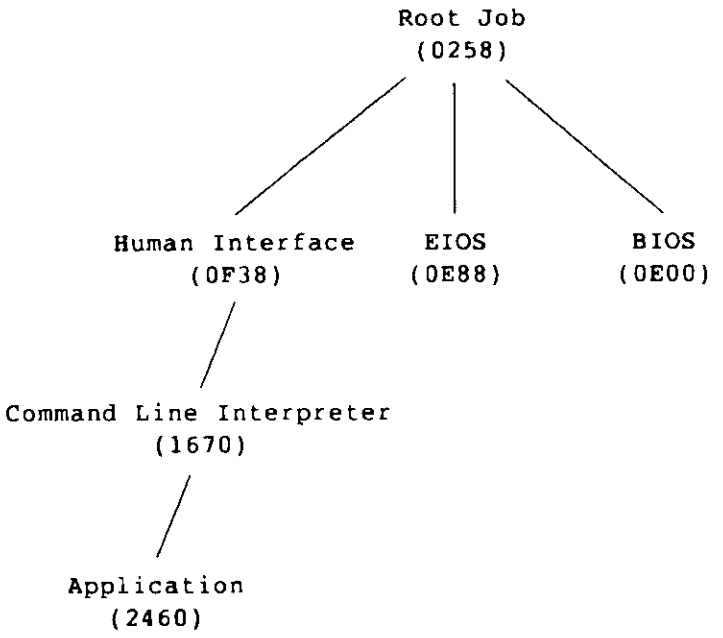


Figure 2-7. iRMX® II Job Tree

If you want to display the descendant jobs of "0E88", enter the following command:

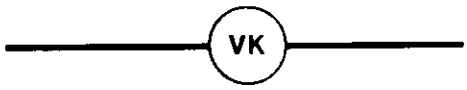
. .

The System Debugger displays the following:

```
        iRMX® II Job Tree

    0E88
    0E00
    0F38
        1670
            2460
```

Note that the tokens for all jobs at the same level as the specified token (0E00 and 0F38), and their descendants (1670 and 2460), are also displayed.

The VK command displays the tokens for tasks in the ready and sleeping states.



x-461

PARAMETERS

This command has no parameters.

DESCRIPTION

The VK command displays the tokens for tasks that are ready and asleep, in the format shown in Figure 2-8.

```
Ready tasks:        xxxx    xxxx    ...

Sleeping tasks:     xxxx    xxxx    ...
```

Figure 2-8. Format of VK Output

The fields in Figure 2-8 show the following:

Ready tasks	The tokens for all tasks in the ready state. The first token in this list represents the running task.
Sleeping tasks	The tokens for all tasks in the sleeping state.

ERROR MESSAGES

Syntax Error	An error was made in entering the command.
Ready tasks: Can't locate	The system is corrupted.
Sleeping tasks: Can't locate	The most common reason for this type of error is not initializing the Nucleus. To recover from this error, reinitialize the system.

EXAMPLE

To display a list of all the ready and sleeping tasks in your system, enter the following command:

```
..
```

The System Debugger responds with the following:

```
Ready tasks:    2F00

Sleeping tasks: 26F0   2588   26B8   2200   21B0   2090   25E8   2050
                2020   1FF8   2698   2238   2118   2668   2638   2768
                20D0   0300
```

The VO command displays the tokens for the objects in the specified job.

x 462

PARAMETER

job token	The token of the job for which you want to display objects.

DESCRIPTION

The VO command lists the tokens for a job's child jobs, tasks, mailboxes, semaphores, regions, segments, extensions, composites, and buffer pools in the format shown in Figure 2-9.

```
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    ...
Buffer Pools:   xxxx    xxxx    xxxx    ...
```

Figure 2-9. Format of VO Output

The fields in Figure 2-9 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 in the job. An "o" following a mailbox token means that one or more objects are queued at the mailbox. A "t" 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 "t" 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 "b" (busy) 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. A "s" following a composite signifies a port with a signal waiting. An "m" signifies a port with a message waiting. A "t" signifies a port with a task waiting.
Buffer Pools	The tokens for the buffer pools in the specified job.

ERROR MESSAGES

Syntax Error	No parameter was specified for the command or an error was made in entering the command.
TOKEN is not a Job	A valid token was entered; however, it is not a job token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).

EXAMPLE

If you want to look at the objects in the job having the token "1670", enter the following command:

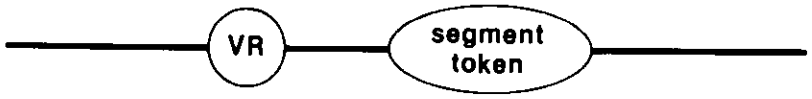
```
. .
```

The System Debugger responds with the following:

```
Child jobs:     2460
Tasks:          1688    1778    17B8    1940    1950    2FF8
Mailboxes:      1720    1728    1738 t  1740 t  1760 t  1768 t
Semaphores:     17A0    17A8 t
Regions:
Segments:       16D8    1750    1958    1960    2FE8    2FC8
Extensions:
Composites:     1690    16F0    1710    1828    1848    1980
Buffer pools:
```

This display shows the tokens for the child jobs, tasks, mailboxes, semaphores, regions, segments, extensions, composites, and buffer pools in the job. It also tells you that tasks are waiting at four mailboxes and one semaphore.

The VR command displays information about the iRMX II Basic I/O System I/O Request/Result Segment (IORS) that corresponds to the segment token you enter.



x 463

PARAMETER

Segment token	The token for a segment containing the IORS you want to display. If this segment is not an IORS, the VR command returns invalid information. To obtain a list of the segment tokens in a job, use the VO command.

DESCRIPTION

The VR command displays the names and values for the fields of a specific IORS. The contents of the IORS reflect the most recent I/O operation in which this IORS was used. Except for ensuring the specified segment is between 45 and 65 bytes long, the System Debugger cannot determine whether the segment contains a valid IORS, so you must ensure that it does. 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 2-10. For more information on I/O Request/Result Segments, see the *Extended iRMX II Basic I/O System User's Guide*.

For more detailed information about the IORS contents, see the *Extended iRMX II Device Drivers User's Guide*.

```
I/O Request Result Segment

Status              xxxx        Unit status     xxxx
Device              xxxx        Unit            xx
Function            xxxxxxx     Subfunction     xxxxxxx
Count               xxxx        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                xxxxx       Cancel ID       xxxx
Connection token    xxxx
```

Figure 2-10. Format of VR Output

The fields in Figure 2-10 are as follows:

Status — The condition code for the I/O operation.

Unit status — Additional status information. The contents of this field are meaningful only when the Status field is set to the E$IO condition (002BH). If the Status field is not set to E$IO, the Unit Status field displays "N/A".

Device — The number of the device for which this I/O request is intended.

Unit — The number of the unit for which this I/O request is intended.

Function — The operation performed by the Basic I/O System. The possible functions are

Function	System Call
Read	RQAREAD
Write	RQAWRITE
Seek	RQASEEK
Special	RQASPECIAL
Att Dev	RQAPHYSICAL$ATTACH$DEVICE
Det Dev	RQAPHYSICAL$DETACH$DEVICE
Open	RQAOPEN
Close	RQACLOSE

If the Function field contains an invalid value, the System Debugger displays the actual 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" from the BIOS RQASPECIAL system call. Possible subfunctions and their descriptions are

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
Set Font	Set character font
Set Bad Info	Set bad track/sector information
Get Bad Info	Get bad track/sector information

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 or logical block where the I/O operation began on the specified device.

Buffer pointer — The address of the buffer the Basic I/O System read from, or wrote to, 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 IORSs 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 (0FFH) and FALSE (00H).
Cancel ID	A word used by device drivers to identify I/O requests that need to be canceled. A value of zero (0) indicates a request that cannot be canceled.
Connection token	The token for the file connection used to issue the request for the I/O operation.

ERROR MESSAGES

Syntax Error	No parameter was specified for the command or an error was made in entering the command.
TOKEN is not a SEGMENT	The token entered is valid but not a segment token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).
SEGMENT wrong size - not an IORS	The specified segment is not between 45 and 65 bytes long, so it is not an I/O Request/Result Segment.

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 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, or you specify a count of zero (0), the number of words in the display depends on the number of parameters for the system call at the CS:IP. Or, in the case when CS:IP is not pointing to a system call, the entire contents of the stack is displayed.

DESCRIPTION

The VS command identifies iRMX II system calls for all iRMX II subsystems (as does the VC command) and interprets the system call parameters on the stack. If the stack does not contain a system call, the VS command displays either the number of stack elements you specify or all of the stack contents, whichever is least. If a parameter is a string, the System Debugger displays the string. For additional system call information, see the appropriate iRMX II Volume 3 system call manual.

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. To move the CS:IP using the iSDM monitor, use the GO (G) command or the EXAMINE/MODIFY REGISTER command (X with CS or IP specified as the 80286 or 80386 register). If you are using the D-MON386 monitor, use only the GO command.

If the instruction is not a CALL instruction, VS displays the contents of the words on the stack and no message. If the instruction is a CALL but not a system call (for example, a PL/M-286 call to a procedure), VS displays the stack contents and a message informing you that the CALL was not a system call.

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 II system call, VS displays the system call and the stack information as shown in Figure 2-11.

```
gate #NNNN
xxxx:xxxx   xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx
xxxx:xxxx   xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx

(subsystem)system call

                    |parameters|
```

Figure 2-11. Format of VS Output

The fields in Figure 2-11 are as follows:

xxxx:xxxx	The contents of the SS:SP (stack memory addresses).
xxxx	Values (tokens) currently on the stack. The number of stack values varies, depending on the number of parameters in the system call.
parameters	The names of the stack values. The parameters correspond to the stack values directly above them. The maximum number of displayed parameters is 24.

The three remaining fields in Figure 2-11 are identical to those in the VC command:

gate #NNNN	The gate number associated with the system call.
(subsystem)	The iRMX II Operating System layer that the system call is part of.
system call	The name of the iRMX II system call.

ERROR MESSAGES

Syntax Error	An error was made in entering the command.
Not a system CALL	The CS:IP is pointing to a CALL instruction that is not an iRMX II 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 belonging to some instruction in the object code for the FAR CALL instruction. The other event is that a software link from user code into iRMX II code has been corrupted. To recover from system corruption, reboot the system.

EXAMPLES

Suppose you determine that the SS:SP is 1906:07CA (using the iSDM Monitor's X command, for example) then use the VS command by entering the following command:

```
..
```

The System Debugger responds with the following:

```
gate #0360
1906:07CA   0B08   1980   1EA8   1980   1980   0000   0B00   1908
1906:07DA   19A0   0B20   0580   1EA8   1EA0   1EE8   0000   0000

(Nucleus) 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 (1980:0B08) to the exception code. Similarly, the "mbox" parameter signifies that the third word (1EA8) is the token for the mailbox being deleted.

Now, suppose that you move the SS:SP to 2906:07D0. If you invoke the VS command by entering the following command:

```
..
```

The System Debugger displays the following stack and a message informing you that the instruction is a CALL instruction but not an iRMX II system call:

```
2906:07D0   2980   2980   0000   0600   2908   29A0   0020   1580
2906:07E0   27C8   27C8   25C8   25C8   25C8   25C8   25C8   25C8

Not a system CALL
```

When an iRMX II system call is executed, its parameters are pushed onto the current stack, and then a CALL instruction is issued with the appropriate stack address. If the call has more parameters than will fit on one line, the System Debugger automatically displays multiple lines of stack values, with corresponding multiple lines of parameter descriptions directly below them.

For example, suppose you use the VS command as follows:

```
..
```

```
gate #0310
27CC:0F9A   0158   20C8   0000   20C8   20C8   0000   0600   17C8
27CC:0FAA   20E8   0028   0000   0000   20C8   00E0   2FF8   2FF8
27CC:0FBA   2608   1A58   1AF8   2608   0000   0000   0000   0000

(Nucleus) 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 with 18 parameters. The names of these parameters are shown between the vertical bars (|). The words on the stack correspond to the parameters directly below them.

The following display shows that the CALL instruction is a Basic I/O System (BIOS) RQAATTACH$FILE system call with five parameters. The "subpath$p" parameter points to a string seven characters long: the word "example."

. .

```
gate #0500
27CC:0F4E   0F88   17C8   25F8   0000   2600   29A0   0000   2600
27CC:0F5E   2608   1C10   2600   1320   26D0   0F78   0DF8   2FF8

(BIOS) 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 RQSRENAME$FILE system call with three parameters. Two of the parameters have strings: the "new$path$p" parameter points to a string four characters long ("XY70"); the "path$p" parameter points to another string four characters long ("temp").

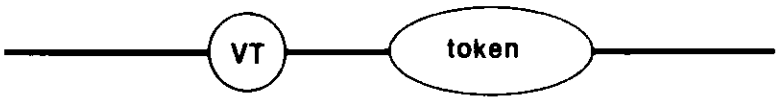
. .

```
gate #06E8

27CC:0F98   0148   20C8   0858   20E8   06A0   20E8   0000   0600
27CC:0FA8   17C8   20E8   0028   1320   0000   20C8   0008   2600

(EIOS) 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 long, the System Debugger displays only the first 50 characters. If the pointer is pointing to a nonreadable segment, the System Debugger does not display the string.

The VT command displays information about the iRMX II object associated with the token you enter.



x-465

PARAMETER

token	The token of the object for which you want to display information.

DESCRIPTION

The VT command determines the type of iRMX II 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 on the type of object.

The following sections are divided into display groups illustrating the display format for these iRMX II objects:

- Jobs
- Tasks
- Mailboxes
- Semaphores
- Regions
- Segments
- Extensions
- Composite objects (six types)
- Buffer Pools

ERROR MESSAGES

Syntax Error	No parameter was specified for the command or an error was made in entering the command.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).

JOB DISPLAY

If the parameter you specify is a valid job token, the System Debugger displays information about the job having that token, as Figure 2-12 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        xxxxx       Pool max       xxxxx   Initial size    xxxxx
Borrowed        xxxxx
```

Byte range	Number chunks	Largest chunk	Total memory
22-44H	xxxxxxxx	xxxxxxxx	xxxxxxxx
44-84H	xxxxxxxx	xxxxxxxx	xxxxxxxx
84-200H	xxxxxxxx	xxxxxxxx	xxxxxxxx
200H-1K	xxxxxxxx	xxxxxxxx	xxxxxxxx
1K-2K	xxxxxxxx	xxxxxxxx	xxxxxxxx
2K-4K	xxxxxxxx	xxxxxxxx	xxxxxxxx
4K-8K	xxxxxxxx	xxxxxxxx	xxxxxxxx
8K-32K	xxxxxxxx	xxxxxxxx	xxxxxxxx
+32K	xxxxxxxx	xxxxxxxx	xxxxxxxx

Figure 2-12. Format of VT Output: Job Display

The fields in Figure 2-12 (from left to right) are as follows:

Current tasks	The number of tasks currently existing in the job. If the Max tasks is not 0FFFFH (no limit), the number of Current tasks is equal to the Current tasks of this job plus all of its children Max tasks.
Max tasks	The maximum number of tasks that can exist in the job simultaneously. This value was set when the job was created.
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 simultaneously. 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 specified by the first parameter when the job was created with the Nucleus RQ$CREATE$JOB system call or the RQE$CREATE$JOB system call.
Entries used	The number of objects currently catalogued in the job's object directory.
Job flags	The job flags parameter specified when the job was created. It contains information the Nucleus needs to create and maintain the job.
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 specified 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.
Borrowed	The current amount (in 16-byte paragraphs) of memory that the job has borrowed from its ancestor(s).
Free Space	All free memory in a job's pool is accounted for, via several double-linked lists. Each list contains a range of chunk sizes. A chunk is a piece of contiguous memory. Column one of the free space table shows the size ranges for the list. Column two shows the number of chunks on each list. Column three displays the largest chunk on each list. Column four shows the total amount of memory on each list.

TASK DISPLAY

The System Debugger displays information about tasks in two different ways. Figure 2-13 shows the display for non-interrupt tasks, and Figure 2-14 shows the display for interrupt tasks.

```
Object type = 2 Task

Static pri      xx          Dynamic pri     xx      Task state      xxxxxxxxx
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      K-saved SS:SP   xxxx:xxxx
```

Figure 2-13. Format of VT Output: Non-Interrupt Task

```
Object type = 2 Task

Static pri      xx          Dynamic pri     xx      Task state      xxxxxxxxx
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      Pending int     xx
Max interrupts  xx          K-saved SS:SP   xxxx:xxxx
```

Figure 2-14. Format of VT Output: Interrupt Task

The fields in Figures 2-13 and 2-14 (from left to right) are as follows:

Static pri — The maximum priority value of the task. This value was set by the max$priority parameter when the task's containing job was created with RQ$CREATE$JOB or RQE$CREATE$JOB.

Dynamic pri — A temporary priority that the Nucleus sometimes assigns to the task to improve system performance. For example, if a higher priority task wants control of a region that belongs to a currently executing lower priority task, the Nucleus assigns the lower priority task a priority equal to that of the higher priority task. This increasing of a task's priority, in this case, improves the overall system performance.

Task state

The state of the task. The twelve possible states, as they are displayed, are

State	Description
ready	task is ready for execution
asleep	task is asleep
susp	task is suspended
aslp/susp	task is both asleep and suspended
deleted	task is being deleted
on exch Q	task is waiting at an exchange
aslp/exch	task is asleep waiting at an exchange
sl/xc/susp	task is asleep and suspended waiting at an exchange
on port Q	task is queued at a port
aslp/port	task is asleep waiting at a port
on trans Q	task is queued at a port on transaction queue
aslp/trans	task is asleep and queued at port on transaction queue

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, this field contains zeros.

Last exchange

The token for the mailbox, region, or semaphore at which the task most recently began to wait.

Except handler

The start address of the job's default exception handler. This value was set either when the task was created with RQ$CREATE$TASK, RQ$CREATE$JOB, RQE$CREATE$JOB, or later with RQSETEXCEPTION$HANDLER.

Except mode	The value that indicates 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 with RQ$CREATE$TASK, RQ$CREATE$JOB, RQE$CREATE$JOB, or later with RQSETEXCEPTION$HANDLER.
Task flags	The task flags parameter used when the task was created with RQ$CREATE$TASK. It contains information the Nucleus needs to create and maintain the job's initial task.
Containing job	The token of the job that this task belongs to.
Interrupt task	Indicates whether this task is an interrupt task. "No" signifies that the task is not an interrupt task. In this case, only the K-saved field follows in the display. (See Figure 2-13.) "Yes" signifies that the task is an interrupt task. In this case, additional fields appear in the display. (See Figure 2-14.)
K-saved SS:SP	The contents of the SS:SP registers when the task last left the ready state.
Int level	The level that the interrupt task services. This level was set when this task called RQSETINTERRUPT.
Master mask	The value associated with the interrupt mask for the master interrupt controller. This value represents the master interrupt levels disabled by the interrupt level that the task services. For example, if the task services master interrupt level 68H, then master levels 6 and 7 are disabled, so the master mask field is 11000000B (=0C0H). For more information about interrupt levels, see the *Extended iRMX II Nucleus User's Guide*.
Slave mask	The value associated with the interrupt mask for a slave interrupt controller. This value represents the slave interrupt levels disabled by the level that the task services. For example, if the task services slave interrupt level 62H, then slave levels 2 through 7 are disabled, so the slave level field is 11111100B (=0FCH). For more information about interrupt levels, see the *Extended iRMX II Nucleus User's Guide*.
Pending int	The number of RQ$SIGNAL$INTERRUPT calls pending for the Int level.
Max interrupts	The maximum number of RQ$SIGNAL$INTERRUPT calls that can be pending for the Int level.

MAILBOX DISPLAY

The System Debugger displays information about mailboxes in three different ways:

- Figure 2-15 shows the display when nothing is queued at the mailbox.
- Figure 2-16 shows the display when tasks are queued at the mailbox.
- Figure 2-17 shows the display when objects are queued at the mailbox.
- Figure 2-18 shows the display when data messages are queued at the mailbox.

```
Object type = 3  Mailbox

Mailbox type        xxxxxx      Task queue head   xxxx
Queue discipline    xxxx        Object queue head 0000
Containing job      xxxx        Object cache depth  xx
```

Figure 2-15. Format of VT Output: Mailbox with No Queue

```
Object type = 3  Mailbox

Mailbox type        xxxxxx      Task queue head   zzzz
Queue discipline    xxxx        Object queue head 0000
Containing job      xxxx        Object cache depth  xx

Task queue          zzzz  xxxx  ...
```

Figure 2-16. Format of VT Output: Mailbox with Task Queue

```
Object type = 3  Mailbox

Mailbox type        xxxxxx      Task queue head   xxxx
Queue discipline    xxxx        Object queue head zzzz
Containing job      xxxx        Object cache depth  xx

Object cache queue      zzzz  xxxx  ...

Object overflow queue   xxxx  xxxx  ...
```

Figure 2-17. Format of VT Output: Mailbox with Object Queue

```
Object type = 3  Mailbox

Mailbox type        xxxxxx      Task queue head   zzzz
Queue discipline    xxxx        Data queue head   xxxx:xxxx
Containing job      xxxx

Data message queue  xxxx:xxxx   xxxx:xxxx   xxxx:xxxx
                    xxxx:xxxx   xxxx:xxxx   ...
```

Figure 2-18. Format of VT Output: Mailbox with Data Message Queue

The fields in Figures 2-15, 2-16, 2-17, and 2-18 are as follows:

Mailbox type	The type of mailbox: object or data. Mailbox type is either Object or Data. The mailbox type is defined when the mailbox is created.
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 the mailbox token.
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 "0000". If the mailbox type is "Data", this field contains "N/A".

Queue discipline	Indicates how tasks are queued at the mailbox. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the mailbox was defined when it was created with RQ$CREATE$MAILBOX. If the System Debugger can't interpret this field, it displays the actual value followed by a space and two question marks.
Object cache depth	The size of the high-performance cache portion of the object queue associated with the mailbox. This size was specified when the mailbox was created with RQ$CREATE$MAILBOX. If the mailbox type is "Data", this field contains "N/A".
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 they are queued. If there are no tasks in the task queue, this field is not displayed.
Object cache queue	A list of tokens for the objects queued in the object cache queue, in the order they are queued. If there are no objects in the object cache queue or the mailbox type is Data, this field is not displayed.
Object overflow queue	A list of tokens for the objects queued in the object overflow queue, in the order they are queued. If there are no objects in the object overflow queue or the mailbox type is Data, this field is not displayed.
Data queue head	The pointer for the first data message at the head of the message queue.
Data message queue	Pointers for the data messages residing at the mailbox.

SEMAPHORE DISPLAY

The System Debugger displays information about semaphores in two ways. The first display appears when no tasks are queued at the semaphore (Figure 2-19), and the second appears when tasks are queued at the semaphore (Figure 2-20).

```
Object type = 4 Semaphore

Task queue head     xxxx    Queue discipline    xxxx
Current value       xxxx    Maximum value       xxxx
Containing job      xxxx
```

Figure 2-19. Format of VT Output: Semaphore with No Queue

```
Object type = 4 Semaphore

Task queue head      xxxx    Queue discipline    xxxx
Current value        xxxx    Maximum value       xxxx
Containing job       xxxx

Task queue       xxxx  xxxx  ...
```

Figure 2-20. Format of VT Output: Semaphore with Task Queue

The fields in Figures 2-19 and 2-20 are as follows:

Task queue head	The token for the task at the head of the queue. If the task queue is empty, this field contains zeros.
Queue discipline	Indicates how tasks are queued at the semaphore. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the semaphore was specified when it was 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 the semaphore belongs to.
Task queue	A list of tokens for the tasks queued at the semaphore, in the order they are queued. If no tasks are queued, this list does not appear.

REGION DISPLAY

If the parameter you supply is a valid token for a region, the System Debugger displays information about the associated region as shown in Figures 2-21 and 2-22.

```
Object type = 5 Region

Entered task         xxxx    Queue discipline        xxxx
Containing job       xxxx
```

Figure 2-21. Format of VT Output: Region with No Queue

```
Object type = 5 Region

Entered task        xxxx    Queue discipline        xxxx
Containing job      xxxx

Task queue      xxxx  xxxx  ...
```

Figure 2-22. Format of VT Output: Region with Task Queue

The fields in Figures 2-21 and 2-22 are as follows:

Entered task	The token for the task currently accessing information guarded by the region.
Queue discipline	Indicates how tasks are queued at the region. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the region was specified when it was created with RQ$CREATE$REGION.
Containing job	The token for the job that the region belongs to.
Task queue	Tokens for the tasks waiting to gain access to data guarded by the region. This line is displayed only if a task is already in the region and another task is waiting.

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 2-23.

```
Object type = 6 Segment

Segment size xxxx              Containing job      xxxx
```

Figure 2-23. Format of VT Output: Segment

The fields in Figure 2-23 are as follows:

Segment size	The number of bytes in this segment. The size of the segment was specified when the segment was created with RQ$CREATE$SEGMENT.
Containing job	The token for the job that the segment belongs to.

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 2-24.

```
Object type = 7 Extension

Extension type      xxxx    Deletion mailbox        xxxx
Containing job      xxxx
```

Figure 2-24. Format of VT Output: Extension Object

The fields in Figure 2-24 are as follows:

Extension type	The type code associated with composite objects licensed by this extension. This code was specified when the extension type was created with RQ$CREATE$EXTENSION. See the *Extended iRMX II Nucleus User's Guide* for more information about extension types.
Deletion mailbox	The token for the deletion mailbox associated with this extension. This mailbox was specified when the extension type was created with RQ$CREATE$EXTENSION.
Containing job	The token for the job that the extension belongs to.

COMPOSITE OBJECT DISPLAY

The VT command displays the following kinds of composite information:

- All composites except those defined in the Basic I/O System (BIOS) and the port connection
- BIOS user objects
- BIOS physical file connections
- BIOS stream file connections

- BIOS named file connections
- BIOS remote file connections
- Port connection

Figure 2-25 shows the format for the display of non-BIOS objects.

```
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 2-25. Format of VT Output: Composite Object Other Than BIOS

The fields in Figure 2-25 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 the composite object belongs to.
Num of entries	The number of component entries in the composite object.
Component list	The list of tokens for the components of the composite.

Figure 2-26 shows the format for the Basic I/O System user object display.

```
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 2-26. Format of VT Output: BIOS User Object Composite

Figure 2-26 uses the composite display described in Figure 2-25 as a base and appends the following fields:

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.

Figure 2-27 shows the format for a (file) connection to a physical file.

```
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       xxxxxx      Open share      xxxxxxx     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     xxxx
Device functs   xxxx        Num dev conn    xxxx        Device name     xxxxxxxxxx
```

Figure 2-27. Format of VT Output: BIOS Physical File Connection

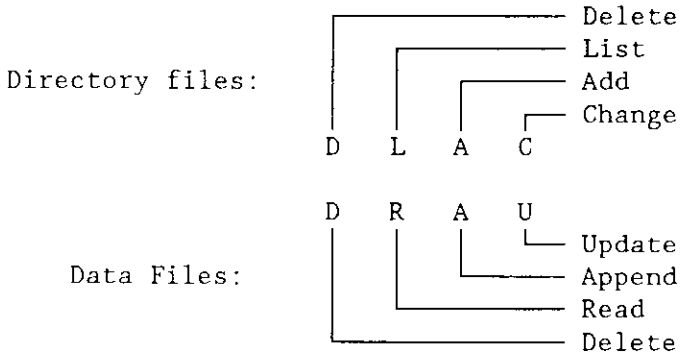
Figure 2-27 uses the composite display described in Figure 2-25 as a base and appends the following fields:

File driver — The BIOS file driver to which this connection is attached. The four possible values are Physical, Stream, Named, and Remote. 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. To determine how the flag is set, convert the hexadecimal value to binary. The following description shows the connection state when a bit (0 is the rightmost bit) is set to 1:

Bit	Condition
0	The connection is being detached
1	The connection is active and can be opened
2	This is a device connection
4	The connection was forcibly detached
3	Reserved
5-7	Reserved

Access — The access rights for this connection. This display uses a single character to represent each access right. If the connection has the access right, the character appears. If the connection does not have an access right, a hyphen (-) appears in the character position. The access rights and the characters that represent them are



Open mode

The mode established when this connection was opened. The possible modes 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 modes are

Share Mode	Description
Private	File cannot be shared
Readers	File can be shared with readers
Writers	File can be shared with writers
ALL	File can be shared with all users
0	Connection is not open

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 IORSs. These IORSs are being saved for the RQ$WAIT$IO system call to use again. This list is empty if zeros appear 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 connections to a device.

Dynamic DUIB
: Indicates whether a Device Unit Information Block (DUIB) was created dynamically when the device associated with this connection was attached.

DUIB pointer
: The address of the DUIB for the device unit containing the file. See the *Extended iRMX II Device Drivers User's Guide* for more information about DUIBs.

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 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 description). The possible modes are

Share Mode	Description
Private	File cannot be shared
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 *Extended iRMX II Basic I/O System User's Guide* for more information about sharing modes for files and connections.

File drivers
: The file drivers that connect the file. If the file can be connected to a given 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
4	Remote 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 where 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-character (or fewer) name of the device where this file resides.

Figure 2-28 shows the format for a (file) connection to a stream file.

```
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     Stream  Conn flags      xx          Access          xxxx
Open mode       xxxxxx  Open share      xxxxxx      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     xxxx
Device functs   xxxx    Num dev conn    xxxx        Device name     Stream
Req queued      xxxx    Queued conn     xxxx        Open conn       xxxx
```

Figure 2-28. Format of VT Output: BIOS Stream File Connection

Figure 2-28 uses the physical display described in Figure 2-27 as a base and appends the following fields:

Req queued	The number of requests currently queued at the stream file.
Queued conn	The number of connections currently queued at the stream file.
Open conn	The number of connections to the stream file currently open.

Figure 2-29 shows the format for a file connection to a named file.

```
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     Named       Conn flags      xx          Access          xxxx
Open mode       xxxxxx      Open share      xxxxxx      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       xxxxxxxxx   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 2-29. Format of VT Output: BIOS Named File Connection

Figure 2-29 uses the physical display described in Figure 2-27 as a base and appends the following fields:

Num of buffers	The number of buffers allocated for blocking and unblocking I/O requests involving the device. A value of zero (0) indicates that the device is not a random-access device.
Fixed update	TRUE or FALSE indicates whether the device uses the fixed update timeout feature. For more information about update timeout, see the *Extended iRMX II Basic I/O System User's Guide*.

Update timeout — The length of the time for the update timeout feature, measured in Nucleus time units. For more information about fixed updating, see the *Extended iRMX II Basic I/O System User's Guide*.

Fnode number — The fnode number of this file. For more information about fnodes, see the *Extended iRMX II 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

If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks.

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 rightmost 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 FFFFH, then the field is displayed as "WORLD". See the *Extended iRMX II Basic I/O System User's Guide* for more information about file ownership.

File/Vol gran — The granularity of the file (in volume granularity units).

Fnode PTR(s) — The addresses of the fnode pointers. See the *Extended iRMX II 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 *Extended iRMX II 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.
This size	The total number of bytes allocated to the file for data.
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.

Figure 2-30 shows the format for a file connection to a remote file.

```
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     Remote  Conn flags      xx          Access          xxxx
Open mode       xxxxxx  Open share      xxxxxx      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
```

Figure 2-30. Format of VT Output: BIOS Remote File Connection

The fields in Figure 2-30 are the same as the fields in Figure 2-27, with the exception of the File driver field, which is "Remote" rather than "Physical."

Figure 2-31 shows the display format for a port having signal protocol type.

```
Object type = 8   Composite

Extension type  xxxx    Extension obj   xxxx    Deletion mbox   xxxx
Containing job  xxxx    Num of entries  xxxx

    T$PORT OBJECT:
Protocol type   Signal  Queue discipline  xxxx  Signal count     xxxx
source id       xxxx

Task queue  xxxx    xxxx
```

Figure 2-31. Format of VT Output: Signal Protocol Port

Figure 2-31 uses the composite display described in Figure 2-23 as a base and appends the following fields:

Protocol type	The message protocol. This value is "Signal" to indicate signal service The type is determined when the port is created through RQ$CREATE$PORT.
Queue discipline	Indicates how tasks are queued at the port. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the port was specified when it was created with RQ$CREATE$PORT. If this field is uninterpretable, the actual BYTE value followed by a space and two question marks appears (??).
Signal count	The number of signals currently waiting to be received at the port.
Source id	The board (agent) identification number for which this port was created to send messages to or receive messages from. This identification number matches the slot number of the remote board. The number is established through the "message$id" field when the port is created using the utility RQ$CREATE$PORT.
Task queue	The tokens for the list of tasks (if any) queued at the port.

Figure 2-32 shows the display format for a port having data transport protocol type.

```
Object type = 8 Composite

Extension type      xxxx    Extension obj       xxxx    Deletion mbox   xxxx
Containing job      xxxx    Num of entries      xxxx

    T$PORT OBJECT:
Protocol type       Data T  Queue discipline    xxxx    Buffer pool     xxxx
Fragmentation       xxx Max Port Transctns  xxxx    Sink port       xxxx
Destination msg id  xxxx    Destination port id xxxx    Source port id  xxxx

Transaction id  xxxx    Task token      xxxx
Transaction id  xxxx    Message pointer xxxx:xxxx

Message queue   xxxx:xxxx   xxxx:xxxx
```

Figure 2-32. Format of VT Output: Data Transport Protocol Port

```
Object type = 8 Composite

Extension type      xxxx    Extension obj       xxxx    Deletion mbox   xxxx
Containing job      xxxx    Num of entries      xxxx

    T$PORT OBJECT:
Protocol type       Data T  Queue discipline    xxxx    Buffer pool     xxxx
Fragmentation       xxx     Max Port Transctns  xxxx    Sink port       xxxx
Destination msg id  xxxx    Destination port id xxxx    Source port id  xxxx

Transaction id  xxxx    Task token      xxxx
Transaction id  xxxx    Message pointer xxxx:xxxx

Task queue      xxxx        xxxx
```

Figure 2-33. Format of VT Output: Data Transport Protocol Port

Figures 2-32 and 2-33 use the composite display described in Figure 2-23 as a base and append the following fields:

Protocol type	The message protocol. This value is "Data T" to indicate Data Transport service The type is determined when the port is created through RQ$CREATE$PORT.

Queue discipline	Indicates how tasks are queued at the port. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the port was specified when it was created with RQ$CREATE$PORT.
Buffer pool	The token of the attached buffer pool (if any). The utility RQ$ATTACH$BUFFER$POOL attaches a buffer pool to a port.
Fragmentation	The fragmentation protocol. This value is either "Yes" if the port can handle message fragmentation, or "No" if the port does not handle message fragmentation. Port fragmentation protocol is defined through the utility RQ$CREATE$PORT.
Max Port Transctns	The maximum number of simultaneous outstanding transactions for the port. This limitation is defined when the port is created using RQ$CREATE$PORT.
Sink port	The token of the sink port (if any) associated with the port. Sink ports are connected to ports through the RQ$ATTACH$PORT utility.
Destination msg id	The host$id portion of the socket identifying the remote port that this port is connected. This value is established through the RQ$CONNECT utility.
Destination port id	The port$id portion of the socket identifying the remote port that this port is connected. This value is established through the RQ$CONNECT utility.
Source port id	The board (agent) identification number for which this port was created to send messages to or receive messages from. The number is established through the "port$id" field when the port is created using the utility RQ$CREATE$PORT.
Transaction id	Outstanding transaction identification numbers at this port.
Task token	The token(s) of the task or tasks with outstanding transactions at this port.
Message pointer	The pointer of the message(s) with outstanding transactions at this port.
Message queue	The list of pointers representing the messages queued at this port. This field appears only if the port has queued messages.

NOTE

In addition to the display forms shown in Figures 2-32 and 2-33, the VT output for a Data Transport protocol port can appear with the following combinations of fields:

- Transaction information with no Message Queue or Task Queue information
- Message Queue information with no Transaction or Task Queue information
- Task Queue information with no Transaction or Message Queue information
- No Transaction, Message Queue, or Task Queue information

BUFFER POOL DISPLAY

If the parameter that you supply is a valid token for a buffer pool, the System Debugger displays information about the buffer pool as shown in Figure 2-34.

```
Object type = 10    Buffer pool

Max Buffers     xxxx    Total buffer count  xxxx    Total size count xxxx
Containing job  xxxx

Buffer pool contents:

Buffer size     xxxx    Buffer count        xxxx
Buffer size     xxxx    Buffer count        xxxx
                .
                .
                .
```

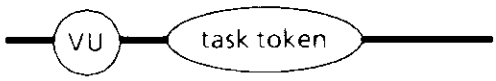
Figure 2-34. Format of VT Output: Buffer Pool

Figure 2-34 display fields are defined as follows:

Max buffers	The total number of buffers allowed in this buffer pool. This maximum value is determined when the buffer pool is created using RQ$CREATE$BUFFER$POOL.
Total buffer count	The number of buffers currently in the buffer pool. This number is equivalent to the number of buffers created in the pool using RQ$CREATE$SEGMENT.

Total size count	The number of different buffer sizes in the buffer pool. The maximum number of different buffer sizes is eight.
Containing job	The token for the job that created this buffer pool.
Buffer size	The available buffer sizes for this buffer pool. These sizes are determined when the individual buffers are created through RQ$CREATE$SEGMENT.
Buffer count	The number of buffers that are of the buffer size displayed in the field directly to the left.

The VU command displays (unwinds) the iRMX II system calls in the stack of the task having the token you enter.



F-0575

PARAMETER

token	The token for the task having the stack 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 II system calls, starting at the top of the stack. For each system call it finds in the stack, it displays

- The return address for the call. This is the address of the next instruction to be executed on behalf of the task after the system call has finished running.
- The VS display with two lines of stack values (or more if required for parameters of the system call), 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 II segment.

The VU command uses internal iRMX II data structures to get some of its information. The data structures are updated immediately after the system call at the top of the task's stack runs to completion. Since the monitor interrupt might come after the system call is completed, but before the data structures are updated, some of the information the VU command uses may be obsolete. Therefore, the first system call the VU command displays may not be valid.

Figure 2-35 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.

If the stack of the indicated task has no system calls, the VU command displays the following message:

```
No system calls on stack
```

```
gate #NNNN

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

(subsystem)  system call

                              |parameters|
```

Figure 2-35. Format of VU Output

The fields in Figure 2-34 are as follows:

gate #NNNN	The gate number associated with the system call.
Return cs:ip	The return address for the system call of this display (yyyy:yyyy).
xxxx:xxxx	The address of the stack portion devoted to this call.
xxxx	Values currently on the stack.
(subsystem)	The iRMX II Operating System layer containing the system call.
system call	The name of the iRMX II system call.
parameters	The parameter names associated with the stack values. The parameters correspond to the stack values directly above them. If one of the parameters is a string, it displays the string contents below the parameters.

ERROR MESSAGES

Syntax Error	An error was made in entering the command.
*** INVALID TASK TOKEN ***	The value entered for the token is not a valid task token.
Stack not an iRMX II segment	The stack of the task is not an iRMX II segment, as is required.
TOKEN is not a TASK	The value entered for the token is valid; however, it is not a task token.

EXAMPLE

This example shows how the VU command responds when system calls are nested. The task for the example has called RQSWRITE$MOVE of the Extended I/O System. RQSWRITE$MOVE has called RQ$A$WRITE of the Basic I/O System. RQAWRITE has called RQ$RECEIVE$MESSAGE to wait for the data transfer to be completed.

Suppose that before the message arrives signaling the completion of the transfer, you enter the System Debugger and invoke the following VU command:

```
..
```

The System Debugger responds by displaying the following:

```
gate #0430

Return cs:ip -09B8:576A
216A:01B2     01C8   216A   01C8   216A   FFFF   1768   1760   1988
216A:01C2     1550   0000   2148   1FF8   1440   2558   2000   2050

(Nucleus) receive message

            |...excep$p....|....resp$p...|.time.|.mbox.|

gate #05B0

Return cs:ip -09D8:08E7
216A:01D4     01E8   216A   1F58   0400   0000   20E8   2098   2088
216A:01E4     1430   2048   01F8   20F8   1400   0218   0000   01F8

(BIOS) write

            |...excep$p...|..mbox.|.count|...buffer$p..|.conn.|

gate #0710

Return cs:ip -09F8:06FA
216A:0218     0020   19F0   0400   0030   19F0   2098   2080   2140
216A:0228     2058   0000   0000   20C8   20C8   20C8   20C8   20C8

(EIOS) write move

            |...excep$p...|..count|...buffer$p...|.conn.|
```

CHAPTER 3
SAMPLE DEBUG SESSION

3.1 INTRODUCTION

This chapter provides a sample PL/M-286 program that was developed on an Intel 310 system running on an iSBC® 286/10 processor board with the iRMX II.3 Operating System. The terminal was a Hazeltine 1510. The code has compiled without errors; however, it does not run. The step-by-step process for using iSDM monitor and System Debugger commands to locate and fix the bug, then to test the corrected code is described in section 3.2. A scenario examining debugging techniques and additional commands is provided in section 3.3.

3.2 SAMPLE PROGRAM

This program includes three tasks: an initialization task (called Init) that creates the other two tasks and a mailbox, and two tasks (called Alphonse and Gaston) that exchange messages via mailboxes. The source code is listed in Figures 3-1, 3-2, and 3-3. For information on compiling and binding this code, see the *Extended iRMX II Programming Techniques Reference Manual*. The following description explains how the program is supposed to work.

The application code runs as a Human Interface program; therefore, the <name of the OBJECT file specified in BND286> is entered at the HI prompt. The task called Init runs first, creating a mailbox it catalogs in the root directory under the name "master." It creates the tasks Alphonse and Gaston then suspends itself.

When Gaston receives control, it gets the token for the mailbox created by Init (by looking up the name "master" in the root job's object directory). It then creates a segment (in which it will place a message) and a response mailbox (to which Alphonse will send a reply). Next it goes into a loop in which it places a message in the segment (after displaying it on the screen), sends the segment to the master mailbox, then waits at the response mailbox for a reply.

When Alphonse receives control, it also gets the token for the mailbox created by Init (by looking up the name in the root job's object directory). It then goes into a loop in which it waits at the mailbox for a message and checks to see if the token it received is a segment. If it is a segment, Alphonse places its own message in the segment (after displaying it on the screen), then sends the segment to the response mailbox. If it isn't a segment, Alphonse drops out of the loop and deletes itself.

By using the two mailboxes, the tasks Alphonse and Gaston are synchronized. Gaston sends a message to the first mailbox and waits at the second one before continuing. Alphonse waits at the first mailbox. When it receives a message, it sends a reply to the second mailbox and waits at the first for another message. This cycle continues for 6 messages.

After sending its sixth message, Gaston drops out of the loop. Instead of sending a segment to the master mailbox, Gaston displays a final message to the screen then sends the task token (the token for the Init task) to the mailbox. When Alphonse receives this token and finds it is not a segment, Alphonse drops out of its loop and deletes itself.

To finish the processing, Gaston causes the Init task to resume processing (remember, the Init task suspended itself earlier). When Init takes over, it deletes both offspring tasks and issues an EXITIOJOB system call to return control to the Human Interface level.

```
compact
init: DO;

DECLARE token                   LITERALLY       'SELECTOR';
DECLARE fifo                    LITERALLY       '0';
DECLARE self                    LITERALLY       '0';
DECLARE task$priority           BYTE;
DECLARE calling$task            TOKEN;
DECLARE calling$tasks$job       TOKEN;
DECLARE master$mbox             TOKEN;
DECLARE status                  WORD;
DECLARE init$task$token         TOKEN;
DECLARE gaston$task$token       TOKEN;
DECLARE alphonse$task$token     TOKEN;
DECLARE alphonse$start$add      POINTER;
DECLARE gaston$start$add        POINTER;
DECLARE gaston$ds               WORD EXTERNAL;
DECLARE alphonse$ds             WORD EXTERNAL;
DECLARE stack$pointer           POINTER;
DECLARE stack$size              WORD;
DECLARE task$flags              WORD;

gaston:
    PROCEDURE EXTERNAL;
END gaston;

alphonse:
    PROCEDURE EXTERNAL;
END alphonse;
```

Figure 3-1. Example PL/M-286 Application (Init)

```
$include(/rmx286/inc/nuclus.ext)
$include(/rmx286/inc/eios.ext)

    calling$tasks$job = SELECTOR$OF(NIL);   /* Directory obj cataloged in */
    calling$task = SELECTOR$OF(NIL);        /* Task whose priority will   */
                                            /* be gotten */
    gaston$start$add = @gaston;             /* Set up start addresses for */
    alphonse$start$add = @alphonse;         /* tasks */
    stack$pointer = NIL;                    /* Values for creating tasks  */
    stack$size = 500;
    task$flags = 0;

    init$task$token = RQ$GET$TASK$TOKENS(   /* Get token for init task    */
        self,
        @status);

    CALL RQ$CATALOG$OBJECT (                /* Catalog task token in      */
        calling$tasks$job,                  /* directory of calling       */
        init$task$token,                    /* task's job */
        @(4,'init'),
        @status);

    master$mbox = RQ$CREATE$MAILBOX (       /* Create mailbox tasks use   */
        fifo,                               /* to pass messages           */
        @status);

    CALL RQ$CATALOG$OBJECT (                /* Catalog mailbox in         */
        calling$tasks$job,                  /* directory of calling       */
        master$mbox,                        /* task's job                 */
        @(6,'master'),
        @status);

    task$priority = RQ$GET$PRIORITY (       /* Get priority of calling    */
        calling$task,                       /* task */
        @status);

    task$priority = task$priority + 1;      /* Pick lower priority for    */
                                            /* new tasks                  */
```

Figure 3-1. Example PL/M-286 Application (Init) (continued)

```
        alphonse$task$token = RQ$CREATE$TASK (  /* Create tasks          */
        task$priority,
        alphonse$start$add,
        SELECTOR$OF(@alphonse$ds),
        stack$pointer,
        stack$size,
        task$flags,
        @status);

    gaston$task$token = RQ$CREATE$TASK (
        task$priority,
        gaston$start$add,
        SELECTOR$OF(@gaston$ds),
        stack$pointer,
        stack$size,
        task$flags,
        @status);

    CALL RQ$SUSPEND$TASK (                  /* Suspend self and let other */
        calling$task,                       /* tasks run                  */
        @status);

    CALL RQ$DELETE$TASK (                   /* Clean up and exit          */
        gaston$task$token,
        @status);

    CALL RQ$DELETE$TASK (
        alphonse$task$token,
        @status);

    CALL RQ$EXIT$IO$JOB (
        0,
        NIL,
        @status);

END;                                        /* Init                       */
```

Figure 3-1. Example PL/M-286 Application (Init) (continued)

```
$compact
alphonse$code: DO;

DECLARE token                          LITERALLY          'SELECTOR';

$include(/rmx286/inc/nuclus.ext)
$include(/rmx286/inc/eios.ext)
$include(/rmx286/inc/hi.ext)

alphonse:
PROCEDURE PUBLIC;

DECLARE CR                             LITERALLY          '13';
DECLARE LF                             LITERALLY          '10';
DECLARE wait$forever                   LITERALLY          '0FFFFH';
DECLARE FOREVER                        LITERALLY          'WHILE 1';
DECLARE calling$tasks$job              TOKEN;
DECLARE master$mbox                    TOKEN;
DECLARE response$mbox                  TOKEN;
DECLARE status                         WORD;
DECLARE type$code                      WORD;
DECLARE time$limit                     WORD;
DECLARE count                          WORD;
DECLARE alphonse$ds                    WORD PUBLIC;
DECLARE seg$token                      TOKEN;
DECLARE seg$size                       WORD;
DECLARE display$message(*)             BYTE               DATA(
    CR,LF, 'After you, Gaston', CR, LF);

DECLARE message BASED seg$token        STRUCTURE(
                                           count          BYTE,
                                           text(25)       BYTE);

    time$limit = 25;                       /* Delay factor for message   */
                                           /* display                    */
    seg$size = 32;                         /* Size of message segment    */
    calling$tasks$job = SELECTOR$OF(NIL);  /* Directory in which to look */
                                           /* up obj                     */
```

Figure 3-2. Example PL/M-286 Application (Alphonse)

```
    master$mbox = RQ$LOOKUP$OBJECT (            /* Look up message     */
        calling$tasks$job,                      /* mailbox             */
        @(6,'master'),
        wait$forever,
        @status);

    DO FOREVER;

        seg$token = RQ$RECEIVE$MESSAGE (        /* Receive response    */
            master$mbox,                        /* from Gaston         */
            wait$forever,
            @response$mbox,
            @status);

        type$code = RQ$GET$TYPE(                /* See what kind of    */
            seg$token,                          /* object it is        */
            @status);

        IF type$code <> 6 THEN                  /* If it isn't a       */
            CALL RQ$EXIT$IO$JOB (               /* segment, exit       */
                0,
                NIL,
                @status);

        message.count = 21;
        CALL MOVB(@display$message, @message.text, size(display$message));

        CALL RQ$C$SEND$CO$RESPONSE (            /* Send message to     */
            NIL,                                /*screen               */
            0,
            @message.count,
            @status);

        CALL RQ$SLEEP(                          /* Wait a while to     */
            time$limit,                         /* give user time to   */
            @status);                           /* see the message     */

        CALL RQ$SEND$MESSAGE (                  /* Send message to     */
            response$mbox,                      /* response mailbox    */
            seg$token,
            SELECTOR$OF(NIL),
            @status);

        END;                                    /* FOREVER             */
    END alphonse;                               /* Alphonse            */
END alphonse$code;
```

Figure 3-2. Example PL/M-286 Application (Alphonse) (continued)

```
$compact
gaston$code: DO;

DECLARE token                       LITERALLY           'SELECTOR';

$include(/rmx286/inc/nuclus.ext)
$include(/rmx286/inc/eios.ext)
$include(/rmx286/inc/hi.ext)

gaston:
PROCEDURE PUBLIC;

DECLARE CR                          LITERALLY           '13';
DECLARE LF                          LITERALLY           '10';
DECLARE fifo                        LITERALLY           '0';
DECLARE wait$forever                LITERALLY           '0FFFFH';
DECLARE parent$task                 TOKEN;
DECLARE calling$tasks$job           TOKEN;
DECLARE master$mbox                 TOKEN;
DECLARE response$mbox               TOKEN;
DECLARE status                      WORD;
DECLARE time$limit                  WORD;
DECLARE count                       WORD;
DECLARE final$count                 WORD;
DECLARE gaston$ds                   WORD PUBLIC;
DECLARE seg$token                   TOKEN;
DECLARE seg$size                    WORD;
DECLARE main$message(*)             BYTE                DATA(
    CR,LF, 'After you, Alphonse', CR, LF);

DECLARE final$message(*)            BYTE                DATA(
    CR,LF, 'If you insist, Alphonse', CR, LF);

DECLARE message BASED seg$token     STRUCTURE(
                                        count           BYTE,
                                        text(27)        BYTE);

    count = 0;                          /* Initialize count          */
    final$count = 6;                    /* Set number of loops       */
    time$limit = 25;                    /* Delay factor for display  */
                                        /* to screen                 */
    seg$size = 32;                      /* Size of message segment   */
    calling$tasks$job = SELECTOR$OF(NIL);   /* Directory in which to look */
                                        /* up object                 */
```

Figure 3-3. Example PL/M-286 Application (Gaston)

```
master$mbox = RQ$LOOKUP$OBJECT (          /* Look up message mailbox   */
    calling$tasks$job,
    @(6,'master'),
    wait$forever,
    @status);

response$mbox = RQ$CREATE$MAILBOX (       /* Create response mailbox   */
    fifo,
    @status);

seg$token = RQ$CREATE$SEGMENT(            /* Create message segment    */
    seg$size,
    @status);

DO WHILE count < final$count;
    message.count = 23;
    CALL MOVW(@main$message, @message.text, SIZE(main$message));

    CALL RQ$C$SEND$CO$RESPONSE (          /* Send message to screen    */
        NIL,
        0,
        @message.count,
        @status);

    CALL RQ$SLEEP(                        /* Wait a while to give user */
        time$limit,                       /* time to see the message   */
        @status);

    CALL RQ$SEND$MESSAGE (                /* Send message to mailbox   */
        master$mbox,
        seg$token,
        response$mbox,
        @status);

    seg$token = RQ$RECEIVE$MESSAGE(       /* Receive response from     */
        response$mbox,                    /* Alphonse                  */
        wait$forever,
        NIL,
        @status);

    count = count + 1;
END;                                      /* WHILE                     */

message.count = 27;
CALL MOVB(@final$message,@message.text,SIZE(final$message));
```

Figure 3-3. Example PL/M-286 Application (Gaston) (continued)

```
        CALL RQ$C$SEND$CO$RESPONSE (        /* Send final message to  */
        NIL,                                /* screen                 */
        0,
        @message.count,
        @status);

    CALL RQ$SEND$MESSAGE (                  /* Send token for mailbox */
        master$mbox,                        /* to mailbox.  This will */
        master$mbox,                        /* stop other task.       */
        SELECTOR$OF(NIL),
        @status);

    parent$task = RQ$LOOKUP$OBJECT(         /* Look up token for      */
        calling$tasks$job,                  /* calling task           */
        @(4,'init'),
        wait$forever,
        @status);

    CALL RQ$RESUME$TASK(                    /* Resume calling task    */
        parent$task,                        /* for cleanup            */
        @status);

    END gaston;                             /* Gaston                 */
END gaston$code;
```

Figure 3-3. Example PL/M-286 Application (Gaston) (continued)

3.3 DEBUGGING THE PROGRAM

Although it's a good idea to include error checking when developing programs, we did not include any in our sample program so that we could demonstrate more features of the System Debugger. The sample program contains one error. We will show two approaches to finding and correcting it using the System Debugger.

The addresses and token values appearing in the following examples are those the system assigned in this debugging session. Most of these values will change from session to session. It's helpful to keep paper and pencil handy to note the various addresses and tokens.

When the iSDM monitor is invoked, both the application code and the operating system code freeze. However, by using iSDM monitor and System Debugger commands you can disassemble and execute the application instructions. Thus, in a debugging session you will move the CS:IP through your code, examining system objects, possibly changing stack or register values. These changes are valid for only one pass through the code. To re-execute the code, kill the current job by using the CLI-restart feature, then re-enter the iSDM monitor by using the Human Interface DEBUG command.

EXAMPLE #1:

When <name of OBJECT file specified in BND286> runs, the system displays the following message:

```
Interrupt 13 at 2C38:0199 General Protection ECODE=0000
..
```

The values 2C38:0199 are where the CS:IP was pointing when the program halted. The protected-mode prompt (..) indicates that we are in the iSDM monitor. However, since the program has been executed, we must re-enter the iSDM monitor to re-execute the code. We can use the CLI-restart feature to return to the Command Line Interpreter. Enter the following command:

```
..
```

The system responds with the Human Interface prompt (-). Next, enter the following command:

```
-
```

The system responds with the following:

```
Interrupt 3 at 2A70:FFFF
..
```

Use the iSDM monitor's GO (G) command to set a breakpoint at the instruction where the program halted (remember the CS:IP value is given in the interrupt message displayed when the program halts). The code segment (CS) value will change each time you re-enter the iSDM monitor, but the instruction pointer (IP) will remain the same. Enter the following command:

```
..
```

To find out where we are in the code, use the iSDM monitor's D (DISPLAY MEMORY/DESCRIPTOR TABLES) command to display a disassembled block of code. Enter the following command:

```
. .
```

The system displays the following code:

```
2500:0199   F2A5        REP     MOVSW
2500:019B   B80000      MOV     AX,0000
2500:019E   8BD0        MOV     DX,AX
2500:01A0   52          PUSH    DX
2500:01A1   50          PUSH    AX
2500:01A2   680000      PUSH    0000
2500:01A5   8E063E00    MOV     ES,[003E]
2500:01A9   B80000      MOV     AX,0000
2500:01AC   06          PUSH    ES
2500:01AD   50          PUSH    AX -
```

The instruction at address 2500:0199 is a MOVE STRING WORD command. The only move word instruction in the sample program is the PL/M-286 MOVW call when Gaston enters the loop after creating the segment. The following display shows this section of code:

```
response$mbox = RQ$CREATE$MAILBOX  (        /* Create response mailbox */
    fifo,
    @status);

seg$token = RQ$CREATE$SEGMENT(              /* Create message segment */
    seg$size,
    @status);

DO WHILE count < final$count;
    message.count = 23;
    CALL MOVW(@main$message, @message.text,  SIZE(main$message));

    CALL RQ$C$SEND$CO$RESPONSE (            /* Send message to screen */
        NIL,
        0,
        @message.count,
        @status);
```

F-0545

Figure 3-4. MOVW in Gaston Code

If displaying the instruction doesn't provide enough information about why the program halted, we can look at the surrounding code by displaying forward or backward from the CS:IP. The comma we specified in the DX command enables us to enter just a comma (,) now to display forward another ten instructions from the current CS:IP. (Displaying backward from the CS:IP is shown in Example #2.)

However, since the instruction where the exception occurred is traceable to the sample code, we know where the program fails. To examine what happens when the system tries to move the message, we'll return to the protected-mode prompt (by entering a carriage return <CR>) and examine register contents before and after MOVSW is executed. Enter the following command:

```
..
```

The system displays the following:

```
AX=0000   CS=2500   IP=0199   FL=0293   RGDT .BASE=002000 .LIMIT=2FFF
BX=0034   SS=2638   SP=01F2   BP=01F2   RIDT .BASE=005000 .LIMIT=03FF
CX=0017   DS=2530   SI=0042  MSW=FFFB
DX=2680   ES=2680   DI=0001   TR=0278   RLDT=02A0
. .
```

To execute the MOVSW instruction, enter the following command:

```
. .
```

The system displays the following:

```
   2500:0199          F2A5          REP MOVSW                    -
```

Enter a comma (,).

The system responds with the following:

```
   Interrupt 13 at 2500:0199 General Protection ECODE=0000
   . .
```

To see how executing this instruction changed register contents, enter the following command:

```
. .
```

The system displays the following:

```
AX=0000   CS=2800   IP=0199    FL=0293   RGDT .BASE=002000 .LIMIT=2FFF
BX=0034   SS=26D8   SP=01F2    BP=01F2   RIDT .BASE=005000 .LIMIT=03FF
CX=0006   DS=28B8   SI=0062   MSW=FFFB
DX=26C0   ES=26C0   DI=0021    TR=0278   RLDT=02A0
. .
```

In the ASM286 Assembly language MOVSW instruction, DS:SI represents the source data is moving from; ES:DI is the destination. (For more information on MOVSW, see the *ASM286 Assembly Language Reference Manual.*) To check the limit of the ES register, enter the following command:

```
. .
```

The system displays the following:

```
GDT (1427T) DSEG BASE=090484 LIMIT=001F P=1 DPL=0 ED=0 W=1 A=1 SR=0000(ES)
```

The LIMIT parameter shows that the segment limit is 1FH (31 decimal). Since the system counts from zero, the limit is 32 decimal which is the value assigned to seg$size in Gaston. The DI register (shown in the previous display) contains 21H (33 decimal), indicating the system was trying to write past the segment limit when the program halted. This fact suggests the PL/M-286 MOVW call should be changed to MOVB. Here we could exit the iSDM monitor, change the PL/M-286 code, then recompile and run it.

However, we can use the iSDM monitor's EXAMINE/MODIFY REGISTERS (X) command to change a register value and the GO (G) command to execute the program. Making changes with the X and S (SUBSTITUTE MEMORY) commands enables us to test code without having to recompile and bind it.

The CX register contains the count of bytes or words moved. If we decrease the count in the CX register to 15 before we execute the MOVSW instruction, we should be able to move all the data. Re-enter the iSDM monitor and set a breakpoint at the MOVSW instruction by entering the following commands:

```
. .
-
. .
```

Set the CX register to 15. Enter the following command:

```
. .
```

Now, execute the rest of the program by entering the following command:

```
. .
```

The system responds with the following:

```
After you, Alphonse

After you, Gaston

Interrupt 13 at 2A70:0199 General Protection ECODE=0000
..
```

Since our change was valid for one pass through the code, the first pass through the Gaston loop worked. The next pass failed. To return to the Command Line Interpreter, enter the following command:

```
..
```

This partially successful run shows that if we reduce the number of words moved, the program works. Therefore, to make a permanent fix, we should change the PL/M-286 MOVW call to MOVB in the sample code, then recompile and bind it.

EXAMPLE #2:

We can also make changes in the disassembled code. Suppose we have run the program for the first time, and the system displayed the following message:

```
Interrupt 13 at 2A70:0199 General Protection ECODE=0000
..
```

Restart the system using the CLI-restart feature as you did in Example #1, then re-enter the iSDM monitor by entering the following command:

```
-
```

Set a breakpoint at the instruction that was executing when the program failed and display a block of disassembled code by entering the following commands:

```
..
..
```

The system displays the following:

```
1258:0199   F2A5     REP      MOVSW
1258:019B   B80000   MOV      AX,0000
1258:019E   8BD0     MOV      DX,AX
1258:01A0   52       PUSH     DX
1258:01A1   50       PUSH     AX
```

To look at the instructions preceding MOVSW, enter the following command:

. .

The system displays the following code:

```
1258:0174   8B063800     MOV      AX,[0038]
1258:0178   3B063A00     CMP      AX,[003A]
1258:017C   7203         JB       A=0181
1258:017E   E97600       JMP      A=01F7
1258:0181   B117         MOV      CL,17
1258:0183   8E063E00     MOV      ES,[003E]
1258:0187   26880E0000   MOV      ES:[0000],CL
1258:018C   B500         MOV      CH,00
1258:018E   8E063E00     MOV      ES,[003E]
1258:0192   BF0100       MOV      DI,0001
1258:0195   BE4200       MOV      SI,0042
1258:0198   FC           CLD
1258:0199   F2A5         REP      MOVSW
1258:019B   B80000       MOV      AX,0000
1258:019E   8BD0         MOV      DX,AX
```

MOVSW is a repetitive move from DS:SI to ES:DI. Looking at the preceding instructions, we see 1258:0181 moves 17H into CL, which is the low-order register of CX. Remember that CX is the count of bytes or words moved. (For more information on the register set, see the *ASM286 Assembly Language Reference Manual*). If we display the ES register contents using "ddt(es) <CR>" as we did in the last example, we can check the limit. Since the limit is 32 (decimal) and the system is trying to write 17H words, the system fails when it tries to write past the segment limit. If we reduce this count we should be able to move the data. We must re-enter the iSDM monitor, then using the iSDM monitor's SUBSTITUTE (S) command, we can change the code at 1258:0181. Semicolons (;) precede the explanations in the following code; enter the information appearing in blue:

```
..
-
..                              ;enter monitor command to substitute
                                 memory at Ip=0181
1258:0181 B1 -                  ;enter a comma to step to the count
1258:0182 17 -                  ;enter the new count
..                              ;re-start code execution
```

The system responds with six iterations of the following:

```
After you, Al
After you, Gaston
.
.
.
```

After six iterations of the previous screen, the monitor displays the following:

```
If you insist, Alphonse
-
```

3.4 VIEWING SYSTEM OBJECTS

Consider that we have a deadlock problem. By looking at system objects at various stages of execution, we can observe how synchronization (or lack of it) is occurring.

We can view any object in a job using the VO command (specifying the job's token) to provide the broad picture of the system state, then the VT command to focus on individual elements. Suppose, we want to view the state of the objects before entering the loop in which Gaston and Alphonse exchange messages. Assume we have stepped through the code, verifying system calls until we located the CS:IP for the Nucleus create$segment system call in Gaston. Re-enter the iSDM monitor and set a breakpoint at this CS:IP by entering the following commands:

```
-Debug <name of OBJECT file specified in BND286>  <CR>
..g,16d  <CR>
```

To get the job token, enter the following command:

```
..
```

The system displays the following:

```
iRMX® II Job Tree

0258                                  Root Job
    0F38                              Human Interface
        1670                          Command Line Interpreter
            2460                      Application
    0E88                              EIOS
    0E00                              BIOS
```

Note that "2460" is the token for the application job. To view objects for this job, enter the following command:

```
..
```

The system displays the following:

```
Child Jobs:
Tasks:          26D0    26F0    1AC8    1900
Mailboxes:      25C0 t  1AB8
Semaphores:
Regions:
Segments:       25B0    25E8    25E0    2650    2528    2480    2478
Extensions:
Composites:     24A0

..
```

At this stage of program execution, two mailboxes exist. The "t" following mailbox 25C0 means one or more tasks are waiting at it (Alphonse was created first and is waiting for a message from Gaston). Examine mailbox 25C0 by entering the following command:

```
..
```

The system responds with the following:

```
Object type = 3     Mailbox

Task queue head          1900    Object queue head       0000
Queue discipline         FIFO    Object cache depth      08
Containing job           2460

Task queue               1900
..
```

Use the System Debugger's VU command to view the waiting task's stack. To unwind the stack, enter the following command:

```
..
```

The system displays the following:

```
gate #0430

Return cs:ip -  1D18:029F
16C8:01E6       0086    1D28    0084    1D28    FFFF    17E0    0000

(Nucleus)receive message

                |...excep$p...|....resp$p...|.time.|.mbox.|

..
```

We can continue to examine objects or set a breakpoint at the return CS:IP. Setting the CS:IP (g, 29f <CR>) in the sample program causes the iSDM monitor to display the following:

```
Interrupt 13 at 21F0:0199 General Protection ECODE=0000
```

This message indicates that the program halts in Gaston and that 21F0:0199 is the instruction executing when it dies.

This chapter has shown two ways to find an error and two ways to make temporary fixes from the System Debugger. The message displayed when the program halts contains the CS:IP of the last instruction executing. If setting the CS:IP at this instruction and displaying the surrounding code doesn't give you enough information about where this point is in your application code, you can use combinations of VJ, VO, VT, VU, and VS to locate the running task. Then set the breakpoint at the CS:IP of the last executing instruction and display code, objects, and registers to determine how the system is executing that instruction.

APPENDIX A
iSDM™ MONITOR COMMANDS

A.1 INTRODUCTION

This appendix briefly describes the iSDM System Debug Monitor commands in alphabetical order. A command directory listing the functional groups and page references precedes the command descriptions. For examples and more detailed information about the commands, see the *iSDM System Debug Monitor User's Guide*.

A.2 COMMAND DIRECTORY

This section provides a brief summary of all iSDM monitor commands listed by functions. Each entry in the following summary contains along with the command name a brief description of the command and a page reference where you can find more information on the command.

Command	Function Performed	Page
PROGRAM LOADING AND EXECUTION		
B	Bootstrap load code from the target system's secondary storage into the target system's memory....	A-3
G	Begin executing application program.....	A-5
L*	Load an 8086 absolute object file or an 80286 object file from a development system into target system memory.....	A-6
N	Execute one or more instructions at a time.....	A-6
R*	Load and execute an 8086 absolute object file or an 80286absolute object file in target system memory...	A-8

Command	Function Performed	Page
I/O PORT INPUT AND OUTPUT COMMANDS		
I	Input and display a byte or word from the specified port	A-5
O	Output a byte or word to the specified port	A-7
BLOCK MANIPULATION		
C	Compare the contents of one block of memory with that of another block	A-4
F	Search the specified block of memory for a sequence of hexadecimal digits	A-5
M	Copy the content of a block of memory to another block of memory	A-6
MEMORY/REGISTER DISPLAY AND MODIFICATION		
D	Display the contents of memory and descriptor table entries	A-4
S	Display and (optionally) modify memory locations and descriptor table entries	A-8
X	Display and/or modify CPU/NPX register or task state segment contents	A-9

Command	Function Performed	Page
MISCELLANEOUS COMMANDS		
E*	Exit the loader program. Return control to the development system	A-4
K*	Echo all console output to a file	A-5
P	Display the base and offset portion of an address or an expression	A-7
Q	Enable Protected Virtual Address Mode (protected mode)	A-7
Y*	Display and define symbol information	A-9

* Command requires an attached development system.

A.3 COMMAND DESCRIPTIONS

This section provides brief descriptions for iSDM monitor commands in an easily referenced alphabetical order. For more information on command parameters, syntax, and options, refer to the *iSDM System Debug Monitor User's Guide*.

A.3.1 B--Bootstrap Load

The B command passes control to the bootstrap loader to load absolute object code from secondary storage into your target system memory. The Bootstrap Loader loads the file into the target system at the memory address specified in the file. After the bootstrap loader finishes loading the file, the code begins executing. To use the B command correctly, you must be operating in real mode.

If either the file you specified or the default file does not exist, the bootstrap loader halts and takes action according to how it is configured.

A.3.2 C--Compare

The C command compares the contents of one block of memory defined by a range with the contents of another block of memory that begins at a destination address. The iSDM monitor expects the blocks to be equal in length. If the iSDM monitor encounters any mismatched bytes, it displays them in the following format:

```
aaaa:bbbb  xx  yy  aaaa:bbbb
```

where "aaaa:bbbb" are the addresses of the bytes that do not match and "xx" and "yy" are the bytes themselves.

A.3.3 D--Display Memory/Descriptor Tables/Disassembled Instructions

The D command is actually three commands in one. You can use it to display the contents of a specified block of memory, the contents of an 80286/80386 descriptor table, or the contents of a specified block of memory in disassembled form. If you are operating in real mode, you cannot display descriptor table entries. However, if you are operating in protected mode, you can use both functions of this command.

A.3.4 E--Exit

The E command enables you to exit the loader program by returning control from the loader program to the development operating system. Upon return, the iSDM monitor loses all symbol information.

When using the E command, you must use it on a line by itself; do not use multiple commands on a line with the E command. Also, your system must include an attached development system before you can use this command.

When you reinvoke the iSDM monitor after exiting the loader program, one of two things happens:

- The iSDM monitor prints either a single or double prompt depending upon whether you were operating in real or protected mode when you exited.
- The iSDM monitor prints its usual sign-on message and re-initializes itself if you reset your target system between the time you exited the loader and the time you reinvoked the iSDM monitor.

A.3.5 F--Find

The F command searches the block of memory you specified to determine if it contains the sequence of hexadecimal digits you chose in the data parameter. Each time the iSDM monitor finds a match, it displays the address of the first matching byte.

A.3.6 G--Go

The G command instructs the iSDM monitor to begin executing your application program. In response to the G command, the iSDM monitor single steps the first instruction, then executes all succeeding instructions at full speed.

Your application program must have at least 12 bytes of stack available for the iSDM monitor to use. If you are operating in protected mode, each task in your program must contain at least 12 bytes of stack at privilege level 0 for the iSDM monitor to use.

With 80286 and 80386 boards, a special situation arises when you execute the G command and you specify a breakpoint address but not a starting address. If the breakpoint is in an interrupt handler and the current CS:IP is at a software interrupt instruction (INT x, INTO, BOUND), the iSDM monitor single steps the interrupt instruction, executing the interrupt handler at full speed and bypassing the breakpoint you set. To get around this 80286/80386 operational anomaly, make sure that the CS:IP is pointing to the (or any) instruction preceding the software interrupt instruction before you execute the G command.

A.3.7 I--Port Input

The I command retrieves and displays a byte or word from the port you specify. Byte and word formats are different. (See the *iSDM System Debug Monitor User's Guide* for byte and word format descriptions).

A.3.8 K--Echo File

The K command copies all console output to a development system file you specify. Repeating the K command without specifying a file causes the iSDM monitor to stop copying console output. Your system must include an attached development system in order to use this command.

A.3.9 L--Load Absolute Object File

The L command loads absolute 8086 or 80286 object files into target system memory. The iSDM monitor loads the data from the file into the memory location that you specified when you used the LOC86 or BLD286 commands. When loading the data, the iSDM monitor discards all previously loaded symbol information and loads the new symbol information, but it retains all user-defined symbols. If the file contains a register initialization record, the iSDM monitor sets the appropriate registers to the values the file specifies. Your system must include an attached development system in order to use this command.

The L command cannot load relocatable modules. If you are operating in real mode, you can load only 8086 absolute object files. If you are operating in protected mode, you can load only 80286 absolute object files.

When you load an 80286 object file, the iSDM monitor initializes the first 40 global descriptor table (GDT) entries for its own use. In addition, the iSDM monitor initializes any uninitialized interrupt descriptor table (IDT) entries. If the access byte is equal to zero, the iSDM monitor assumes that the descriptor table entry is not initialized. Refer to Intel's *Microprocessor and Peripheral Handbook*, *Microsystem Components Handbook*, or *IAPX 286 Operating System Writer's Guide* for more information about the descriptor tables.

A.3.10 M--Move

The M command copies the contents of a block of memory to a memory address you specify.

A.3.11 N--Execute Single Instructions

The N command displays and executes one or more disassembled instructions at a time. Going through your application line-by-line is called "single-stepping." Single-stepping allows you to begin at a CS:IP you specify and check your application for problems in an instruction-by-instruction manner.

Your application program must have at least 12 bytes of stack available for the iSDM monitor to use. If you are operating in protected mode, each task in your program must contain at least 12 bytes of stack at privilege level 0 for the iSDM monitor to use.

When you are single-stepping instructions, you should be aware of some special considerations. See the *iSDM System Debug Monitor User's Guide* for more information about these special considerations when using the N command.

A.3.12 O--Port Output

The O command allows you to enter data (a byte or word) at the console and send it to a port you select.

A.3.13 P--Print

The P command allows you to display either the value of an expression or the value of the base (or selector) and offset portions of an address. The values are displayed on your console terminal screen. The iSDM monitor always displays an address in hexadecimal form. If you enter "P" plus an expression, the iSDM monitor prints the value in hexadecimal. If you enter "PT" or "PS" plus an expression, the iSDM monitor prints the value in decimal or signed decimal form, respectively.

In this command, the comma acting as a separator also causes the iSDM monitor to add a space between the addresses or expressions it displays.

A.3.14 Q--Enable Protection (80286/80386 Only)

The Q command changes the 80286- or 80386-based system from real mode to protected mode. The iSDM monitor displays the following message when you use the Q command:

```
Now in Protected Mode
```

When you invoke this command, the iSDM monitor initializes the entries it needs in the GDT and the IDT. The iSDM monitor then places itself at privilege level zero. If you are already operating in protected mode when you invoke this command, the iSDM monitor re-initializes the GDT and IDT entries. The only way you can return to real mode is to reset the 80286 or 80386 hardware.

A.3.15 R--Load and Go

The R command is a combination of the Load command (L) and the Go command (G). This command loads an absolute object file from a development system into target system memory then executes this program. This command causes the iSDM monitor to discard all previously loaded symbol information and load new symbol information; however, the iSDM monitor retains all user-defined symbols. Your system must include an attached development system in order to use this command.

The iSDM monitor loads the data from the file into the memory location that you specified when you used the LOC86 or BLD286 commands. If the file contains a register initialization record, the iSDM monitor sets the appropriate registers to the values the file specifies.

The R command cannot load relocatable modules. If you are operating in real-addressing mode, you can load only 8086 absolute object files. If you are operating in protected mode, you can load only 80286 bootloadable (absolute) files.

When you load an 80286 object file, the iSDM monitor initializes the first 40 global descriptor table (GDT) entries for its own use. In addition, the iSDM monitor initializes any uninitialized interrupt descriptor table (IDT) entries. Refer to Intel's *Microprocessor and Peripheral Handbook*, *Microsystem Components Handbook*, or *iAPX 286 Operating System Writer's Guide* for more information about the 80286 component's descriptor tables.

After the iSDM monitor loads the file and sets the appropriate registers to the values the file specifies, it begins to execute the program at the location specified by the CS and IP registers.

Your application program must have at least 12 bytes of stack available for the iSDM monitor to use. If you are operating in protected mode, each task in your program must contain at least 12 bytes of stack at privilege level 0 for the iSDM monitor to use.

A.3.16 S--Substitute Memory/Descriptor Table Entry

The S command is actually two commands in one. You can use it to display and (optionally) modify either the contents of memory or the contents of descriptor table entries. If you are operating in real mode, you cannot display and modify descriptor table entries. However, if you are operating in protected mode, you can use both functions of this command.

If you enter the S command without an equal sign (=), the iSDM monitor displays a special hyphen (-) prompt. Then, it waits for you to enter either

- A continuation comma instructing the iSDM monitor to display the next memory location.
- A single expression or a list of expressions separated by slashes (/). By entering an expression (or expressions), you instruct the iSDM monitor to substitute these values in place of those already in the memory location you specified.

The iSDM monitor continues to issue hyphen prompts until you enter a carriage return.

A.3.17 X--Examine/Modify Registers

The X command allows you to examine and (optionally) modify the contents of your system's NPX and microprocessor registers.

If you use the X command with no parameters, the iSDM monitor displays all of the 8086, 286, and 386 registers.

If you use both the register name and an expression, (for example, CS = XXXX), the value you entered (XXXX) is placed in the specified register.

You can use the X command to set the 8086 family and NPX registers and the task state segment contents to any value. If you used any invalid values, the iSDM monitor reports them when you execute the application program.

A.3.18 Y--Symbols (80286 or 80386 Only)

The Y command allows you to display and define symbol information generated by 80286 translators. If you use the Y command with no parameters, the iSDM monitor displays all the symbols stored in the current domain module or in all modules if you set no domain. You can also choose to have the iSDM monitor display the symbols and their values in a particular module or you can use this command to define your own symbols. To use this command, you must be operating in protected mode, with an attached development system.

APPENDIX B
D-MON386 COMMANDS

B.1 INTRODUCTION

This appendix briefly describes the 80386 Debug Monitor (D-MON386) commands in alphabetical order. A command directory listing the functional groups and page references precedes the command descriptions. For examples and more detailed information about the commands, see the *D-MON386 Debug Monitor for the 80386 User's Guide*.

B.2 ENTERING COMMANDS

To enter D-MON386 commands, follow the guidelines below:

- Terminate a command line by pressing the ENTER key or the RETURN (<CR>) key. A command line can consist of one or more commands.
- Separate multiple commands on a single line using a semicolon (;).
- Continue commands from one line to another by entering the slash (/) just before terminating the line with the ENTER key or RETURN key.
- Enter commands using upper or lower case characters.
- Use CTRL-C (pressing the control key down while at the same time pressing the C key) to abort a command being constructed on the command line.

B.3 COMMAND DIRECTORY

This section provides a brief summary of all D-MON386 commands listed by functions. Each entry in the following summary contains along with the command name a brief description of the command and a page reference where you can find more information on the command.

Command	Function Performed	Page
BLOCK		
COUNT/ENDCOUNT	Provides monitor command control structures. These structures enable you to enter and repeat execution of several monitor commands	B-5
CONTROL VARIABLES		
BASE	Display or set the base number system to to either binary, octal, decimal, or hexadecimal	B-5
N0-N9	Display or set scratch registers zero through nine	B-8
$	Display or set the current execution point	B-5
EXPRESSION DISPLAY		
EVAL	Evaluates an expression and displays the results	B-6
EXECUTION ENVIRONMENT		
GO	Controls high-level execution environment	B-7
ISTEP	Enables single-step execution	B-8
SWBREAK	Displays and sets software code breaks	B-9
SWREMOVE	Removes software code breaks	B-10

Command	Function Performed	Page
DESCRIPTOR TABLE ACCESS		
GDT	Displays the Global Descriptor Table or specific entries	B-7
LDT	Displays the Local Descriptor Table or specific entries	B-8
IDT	Displays the Interrupt Descriptor Table or specific entries	B-7
DT	Displays the Global or Local Descriptor tables	B-6
MEMORY ACCESS		
ASM	Disassembles memory as 80386 assembler mnemonics	B-5
BYTE	Reads or writes bytes of memory	B-5
DWORD	Reads or writes double words of memory	B-6
INTn	Reads or writes 1-, 2-, or 4-byte integers in memory	B-7
ORDn	Reads or writes 1-, 2-, or 4-byte ordinals in memory	B-8
USE	Initializes the default for disassembling code to 16-bit or 32-bit	B-10
WORD	Reads or writes words of memory	B-10
PAGE TABLE ACCESS		
PD	Displays the Page Table Directory or page table entries	B-8

Command	Function Performed	Page
PORT I/O		
DPORT	Reads or writes 32-bit ports	B-6
PORT	Reads or writes 8-bit ports	B-9
WPORT	Reads or writes 16-bit ports	B-11
REGISTER ACCESS		
CREGS	Displays the control registers	B-6
FLAGS	Displays the lower 16 bits of the EFLAGS register in mnemonic form	B-6
Register-name	Displays or modifies individual registers	B-9
REGS	Displays a set of selected registers as a group	B-9
SREGS	Displays the segment registers as a group	B-9
TASK STATE SEGMENT ACCESS		
TSS	Displays the contents of a task state segment	B-10
USER AID		
B	Executes a real mode interface program	B-5
HELP	Displays the help screen	B-7
HOST	Provides the capability for operation with PMON host software	B-7
VERSION	Displays the version of D-MON386	B-10

B.4 COMMAND DESCRIPTIONS

This section provides brief descriptions for D-MON386 commands in an easily referenced alphabetical order. For on-line syntax help, refer to the HELP command. For more information on command parameters, syntax, and options, refer to the *D-MON386 Debug Monitor for the 80386 User's Guide*.

B.4.1 $

This command displays or modifies the current execution point via the execution address register (CS:EIP). The contents of CS:EIP determine which ASM386 statement executes next. Entering $ by itself displays the current contents of CS:EIP.

B.4.2 ASM

This command disassembles code into ASM386 opcode mnemonics. Using this command and the addresses you supply with it, you can disassemble from one to several lines of code. Disassembled code appears on the screen in column form. Each row of columns contains an address, a hexadecimal object value, an opcode mnemonic, any operands, and comments appended to the operands.

B.4.3 B

This command invokes a user-supplied real mode interface program. The B command is intended primarily for including a bootstrap loader program.

B.4.4 Base

This command displays or modifies the number base. Available number bases include binary, octal, decimal, and hexadecimal. The hexadecimal base is the monitor default base. Entering BASE by itself displays the current base. Entering BASE followed by an expression that evaluates to 2, 8, 10, or 16 (all decimal numbers) sets the base to binary, octal, decimal, or hexadecimal, respectively.

B.4.5 Byte

This command displays or modifies partitions of memory using a byte format. You can specify the partition as a single byte or a range of bytes. Entering the command BYTE followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command BYTE as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.6 Count/Endcount

This command executes groups of D-MON386 commands in a specified order for a specified number of times. After entering COUNT expr, simply enter in commands you wish to execute. After entering ENDCOUNT, one iteration of the commands will have already been executed. The entire group of commands then continues to execute for expr-1 number of times.

B.4.7 Cregs

This command displays the contents of the control registers and the EFLAGS register when the processor is in real mode. If the processor is in protected mode, the CREGS command also displays the system address registers TR and LDTR. The display appears using a hexadecimal number base.

B.4.8 Dport

This command reads or writes a 32-bit port. Entering DPORT with the physical input/output address space as a 16-bit unsigned quantity causes the specified port to be read and the contents to appear on the screen. If you supply an expression to the right of the equal sign when entering this command, the addressed port is written with the value the expression equals.

B.4.9 DT

This command displays descriptors from either the LDT or the GDT depending upon the index supplied with the command.

B.4.10 Dword

This command displays or modifies partitions of memory using a double word format. You can display a specific double word or a range of double words by entering DWORD followed by the single address or the range of addresses. Entering the DWORD command as an equation causes the partition of memory specified on the left-hand side of the equation to be replaced with the contents of memory or value of the right-hand side of the equation.

B.4.11 Eval

This command evaluates the expression entered in after the keyword EVAL. The results of the expression appear on the screen in binary, octal, decimal, hexadecimal, and ASCII formats.

B.4.12 Flags

This command displays the contents of the lower 16 bits of the EFLAGS register. The display appears in a mnemonic form. The presence of a mnemonic indicates a flag is set. The absence of a mnemonic in the display indicates a flag is not set.

B.4.13 GDT

This command displays the entire Global Descriptor Table (GDT) or individual GDT descriptors. Entering the keyword GDT by itself causes the entire GDT to appear. Entering GDT followed by an index expression causes a specific descriptor to appear.

B.4.14 Go

This command supplies high-level execution control. Use of the GO command enables you to begin and end program execution using specific points in the application. You can also clear and specify break conditions using the GO command.

B.4.15 Help

This command displays the major D-MON386 commands along with their general syntax. For examples and more detailed information about the commands, see the *D-MON386 Debug Monitor for the 80386 User's Guide*.

B.4.16 Host

This command provides the capability for operation with PMON host software. When entering this command, be sure to press only the ENTER key or a carriage return <CR> immediately after HOST.

B.4.17 IDT

This command displays the entire Interrupt Descriptor Table (IDT) or individual IDT descriptors. Entering the keyword IDT causes the entire IDT to appear. Entering IDT followed by an index causes a specific descriptor from the IDT to appear.

B.4.18 INTn

This command displays or modifies partitions of memory using an integer format. When entering the command, you can substitute the numbers 1, 2, or 4 for n. Thus, the integer type(s) referenced in memory are either 1-, 2-, or 4-byte integers. You can specify the partition as a single INTn value or a range of INTn values. Entering the command INTn followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command INTn as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.18 Istep

This command performs single-step execution. You can use this command to single-step through the executable code from one to 255 executable statements. ISTEP also provides the capability to begin execution from a point other than the current execution point.

B.4.19 LDT

This command displays the entire Local Descriptor Table (LDT) or individual LDT descriptors. Entering the keyword LDT causes the entire LDT to appear. Entering LDT followed by an index causes a specific descriptor from the LDT to appear.

B.4.20 N0-N9

This command displays or alters scratch registers zero through nine. Entering Nn (where n is a number 0 through 9) by itself causes the value of the appropriate register to appear on the screen. You can enter Nn followed by an equal sign and an expression to alter the contents of the appropriate scratch register.

B.4.21 ORDn

This command displays or modifies partitions of memory using an ordinal format. When entering the command, you can substitute the numbers 1, 2, or 4 for n. Thus, the ordinal type(s) referenced in memory are either 1-, 2-, or 4-byte ordinals. You can specify the partition as a single ORDn value or a range of ORDn values. Entering the command ORDn followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command ORDn as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.22 PD

This command examines the Page Table Directory and page tables. When paging is enabled, the 80386 uses two levels of tables to translate a linear address into a physical address: the Page Table Directory and the page tables themselves. Entering the PD command by itself causes the entire 4K Page Table Directory to scroll to the screen. You can, however, supply an index with the PD command to view a particular directory entry within the Page Table Directory. Also, you can use the additional .PT option with an index to view a particular page table entry.

B.4.23 Port

This command reads or writes a 8-bit port. Entering PORT with the physical input/output address space as a 16-bit unsigned quantity causes the specified port to be read and the contents to appear on the screen. If you supply an expression to the right of the equal sign when entering this command, the addressed port is written with the value the expression equals.

B.4.24 Register-name

D-MON386 enables you to display or alter the contents of 80386 registers. To gain register access, enter the name of the register Entering the name of the register only causes the contents of the register to appear on the screen. Entering the name of the register followed by an equal sign and a valid expression causes the contents of the register to be written with the value of the expression. For a complete list of register names, refer to the *u-MON386 Debug Monitor for the 80386 User's Guide*.

NOTE

Register modification is dependent on the current processor protection model. You cannot modify protected registers.

B.4.25 Regs

This command displays the contents of a set of registers as a group. The register set depends on which mode the processor is currently operating under (real or protected). The display is always in hexadecimal, and it provides less detail for the segment and control registers than the command that are specifically designed for those groups of registers, that is SREGS and CREGS, respectively.

B.4.26 Sregs

This command displays, in hexadecimal, the contents of the segment registers (CS, DS, SS, ES, FS, and GS).

B.4.27 Swbreak

This command displays or sets code patch breaks. Entering SWBREAK by itself causes all current software break definitions to appear. If you enter SWBREAK followed by an equal sign and one or more addresses, the command sets a software break at the specified address or addresses.

NOTE

When specifying software break addresses, the address must be able to be written, present in physical memory, and on an instruction boundary. A maximum of 16 software breaks may be in effect at one time.

B.4.28 Swremove

This command removes all or selected code patch breaks. Entering this command followed by ALL removes all current software breaks. If you supply one or more addresses with the command, the software breaks at those addresses alone are removed.

B.4.29 TSS

This command displays the contents of a task state segment. TSS supports both 80386 and 80286 task state segments. Task state segments appear using the component names.

B.4.30 Use

This command specifies the default (16-bit or 32-bit code) for disassembling code from physical or linear addresses. When entering the command, the expression to the right of the equal sign must evaluate to either 16 or 32 (decimal).

B.4.31 Version

This command displays the version number of the D-MON386 software you are using.

B.4.32 Word

This command displays or modifies partitions of memory using a word format. You can specify the partition as a single word or a range of words. Entering the command WORD followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command WORD as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.33 Wport

This command reads or writes a 16-bit port. Entering WPORT with the physical input/output address space as a 16-bit unsigned quantity causes the specified port to be read and the contents to appear on the screen. If you supply an expression to the right of the equal sign when entering this command, the addressed port is written with the value the expression equals.

Intel®

INDEX

A

Altering descriptor table entries A-8
Altering memory contents A-8, B-5, 6, 7, 8, 10
Altering register contents 3-14, A-9, B-8, 9

B

Bootloading from the monitor A-3, B-5
Bootstrap Loader DEBUG switch 1-3
Breakpoints 1-3, 3-10, B-9, 10

C

Changing current instruction pointer B-5
Changing disassembled code 3-15, 16
Changing descriptor table entry contents A-8
Changing memory contents A-8, B-5, 6, 7, 8, 10
Changing modes A-7
Changing register contents 3-14, A-9, B-8, 9
CLI-restart 1-3, 2-3, 3-10
Code blocks, displaying 3-12, 16
Commands
- D-MON386 B-1
- Directory 2-4
- iSDM™ A-1
- Overview 1-3
- Syntax for debugger 1-3
- Token validity 2-1
- VB 2-5
- VC 2-9
- VD 2-12
- VF 2-14
- VH 2-16
- VJ 2-18, 3-18
- VK 2-22
- VO 2-24, 3-17, 18
- VR 2-27

Commands (cont.)
VS 2-31
VT 2-36, 3-17, 18
VU 2-62, 3-19
Comparing blocks of memory A-4
Configuration 1-2
Contents of the stack 2-31
Conventions iv, 2-1
Copying blocks of memory A-6
Current instruction, displaying 3-11, B-5

D

D-MON386 monitor command directory B-2
D-MON386 monitor command overview B-1, 5
D-MON386 monitor commands
$ B-5
Asm B-5
B B-5
Base B-5
Byte B-5
Count/Endcount B-5
Cregs B-6
Dport B-6
DT B-6
Dword B-6
Eval B-6
Flags B-6
GDT B-7
Go B-7
Help B-7
Host B-7
IDT B-7
INTn B-7
Istep B-8
LDT B-8
N0-N9 B-8
ORDn B-8
PD B-8
Port B-9
Register name B-9
Regs B-9
Sregs B-9
Swbreak B-9

D-MON386 monitor commands (cont.)
Swremove B-10
Syntax B-1
TSS B-10
Use B-10
Version B-10
Word B-10
Wport B-11
Deadlock 3-17
DEBUG command 1-2, 3-10
Debug session, sample 3-1
Descriptor tables, displaying A-4, B-6, 7, 8
Determining the base and offset of an address A-7
Disassembled code, displaying 3-15, A-4, B-5, 10
Displaying blocks of code 3-12, A-4
Displaying symbol information A-9
Displaying the number base B-5
DUIB information, displaying 2-5

E

Echoing console output A-5
ES register limit, checking 3-14
Examining a mailbox 3-18
Examining page table directory and tables using D-MON386 B-8
Examining register contents 3-12, 13, B-6, 9
Examining stack contents 3-19
Example debug session 3-1
Executing a program 3-14, 15, A-5, 8, B-7
Executing a single line of code 3-13, A-6, B-8
Exiting the monitor A-4
Expression evaluation A-7, B-6

F

Finding text A-5

G

GDT slots, displaying free amount 2-14
Getting help 2-16, B-7

H

Hardware/Software requirements 1-2
Help 2-16, B-7

I

I/O Result Segment (IORS) 2-27
Identifying system call parameters on the stack 2-31
Interpreting system call parameters on the stack 2-31
Invocation 1-2, 3-10
IORS, displaying 2-27
ISDM™ monitor command directory A-1
ISDM™ monitor command overview A-1
ISDM™ monitor commands
B - bootstrap load A-3
C - compare A-4
D - Display 3-11, A-4
E - exit A-4
F - find A-5
G - go 3-10, 14, 15, A-5
I - port input A-5
K - echo file A-5
L - load A-6
M - move A-6
N - single instruction execution 3-13, A-6
O - port output A-7
P - print A-7
Q - enable protection A-7
R - load and go A-8
S - substitute A-8
X - examine/modify 3-12, 13, 14, A-9
Y - symbols A-9

J

Job and descendent job tokens, displaying 2-18

L

Loading object files A-6, 8
Locating running tasks 3-20
Looping within D-MON386 B-5

M

Mailbox examination 3-18
Manual Overview iii
Modifying the number base B-5
Mode changing A-7
Monitor 1-1

Monitor commands
iSDM™ A-1
D-MON386 B-1
Moving blocks of memory A-6

O

Object directory, displaying 2-12
Objects, displaying 2-24, 3-18

P

Ports
Displaying data A-5, B-6, 9, 11
Entering data A-7, B-6, 9, 11
Product overview iii, 1-1
Program code execution 3-13

Q

Quitting the debugger 1-4, A-4

R

Re-entering the iSDM™ monitor 3-10, 15
Reader level iii
Redirecting console output A-5
Removing Breakpoints with D-MON386 B-10
Register contents, examining 3-12, 13
Returning to your application 1-4

S

Sample debug session 3-1
Searching for text A-5
Setting breakpoints 1-3, 3-10, B-9
Single-step execution A-6, B-8
Stack contents 2-31, 3-19
Starting the debugger 1-2
Strings, display limitations 2-35
Support 1-2
Symbol information, displaying A-9
Syntax for D-MON386 commands B-1

Syntax for debugger commands 1-3, 2-2
System call information, displaying 2-9
System call parameters on the stack, displaying 2-31
System requirements 1-2

T

Task system calls, displaying 2-62
Task tokens, displaying 2-22
Tokens, displaying 2-36, 3-18

U

Using PMON host software with D-MON386 B-7
Using the debugger 1-3, 3-9

V

VB command 2-5
VC command 2-9
VD command 2-12
Version number of D-MON386, displaying B-10
VF command 2-14
VH command 2-16
VJ command 2-18
VK command 2-22
VO command 2-24, 3-17
VR command 2-27
VS command 2-31
VT command
  Buffer pool display 2-60
  Composite object display 2-47
  Extension object display 2-47
  Job display 2-37
  Mailbox display 2-42, 3-18
  Region display 2-45
  Segment display 2-46
  Semaphore display 2-44
  Task display 2-39
VT command 2-36
VU command 2-62

W

Warm-start 1-3, 2-3

intel

EXTENDED iRMX® II DISK VERIFICATION UTILITY REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

Intel®

PREFACE

INTRODUCTION

The iRMX II Disk Verification Utility is a software tool that runs as a Human Interface command verifying and modifying the data structures of iRMX named and physical volumes.

This manual describes the utility invocation and contains detailed descriptions of all utility commands. It also documents the iRMX II capability of backing up and restoring volume file descriptor nodes (fnodes).

In addition, the manual describes the structure of iRMX named volumes as users must be familiar with volume structure to use the full capabilities of the Disk Verification Utility.

READER LEVEL

This manual is intended for programmers who have an understanding of the operating system, and particularly the Basic I/O System and Human Interface layers. To use this manual effectively, programmers should be familiar with iRMX volume structure. Appendix A provides a brief review of iRMX named volume structure. However, this is intended as a reference and not as a tutorial.

MANUAL OVERVIEW

This manual is organized as follows:

Chapter 1	This chapter describes two ways of invoking the Utility: single-command mode or interactive mode. It explains single-command mode and how to interpret output and error messages from the single-command verification. It also describes the invocation in interactive mode and the interactive mode error messages. Commands for the interactive mode are explained in Chapter 2.

Chapter 2	This chapter contains detailed descriptions of the Disk Verification Utility commands. The commands are discussed in alphabetical order. When verifying and modifying volumes, you should refer to this chapter for specific information about the format and parameters of the commands.
Chapter 3	This chapter explains the fnode backup and restore feature in detail. This feature provides a limited mechanism for attempting to recover data when the volume label or the fnode file has been damaged.
Appendix A	This appendix provides information on the format of iRMX named volumes. It includes details of the volume label and fnode file, differences between long and short files, and format information specific to diskettes. Programmers should be familiar with this information before attempting to modify a volume.

CONVENTIONS

This manual uses the following conventions:

- Information appearing as UPPERCASE characters when shown in keyboard examples must be entered or coded exactly as shown. You may, however, mix lower and uppercase characters when entering the text.
- Fields appearing as lowercase characters within angle brackets (< >) when shown in keyboard examples indicate variable information. You must enter an appropriate value or symbol for variable fields.
- User input appears in one of the following forms:

```
as bolded text within a screen
```

- text is used to indicate the first occurrence of each command described in Chapter 2; subsequent occurrences are printed in black ink.
- The terms "iRMX II" and "Operating System" refer to the Extended iRMX II Operating System.
- The term "iRMX I" refers to the iRMX I Operating System (iRMX 86 Operating System).
- All numbers unless otherwise stated are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

Intel®

CONTENTS

CHAPTER 1
INVOKING DISKVERIFY **PAGE**

1.1 Introduction 1-1
1.2 Invocation 1-2
1.3 Output 1-5
1.4 Invocation Error Messages 1-6

CHAPTER 2
DISKVERIRY COMMANDS **PAGE**

2.1 Introduction 2-1
2.2 Command Syntax 2-1
2.3 Command Names 2-2
2.4 Parameters 2-3
2.5 Input Radices 2-3
2.6 Aborting Diskverify Commands 2-4
2.7 Diskverify Error Messages 2-4
2.8 Command Dictionary 2-5
ALLOCATE 2-8
BACKUPFNODES 2-11
DISK 2-13
DISPLAYBYTE 2-16
DISPLAYWORD 2-18
DISPLAYDIRECTORY 2-20
DISPLAYFNODE 2-23
DISPLAYSAVEFNODE 2-27
DISPLAYNEXTBLOCK 2-28
DISPLAYPREVIOUSBLOCK 2-29
EDITFNODE 2-30
EDITSAVEFNODE 2-33
EXIT 2-34
FIX 2-35
FREE 2-38
GETBADTRACKINFO 2-41
HELP 2-43
LISTBADBLOCKS 2-44

CHAPTER 2 (continued) **PAGE**

MISCELLANEOUS COMMANDS 2-46
ADD 2-46
ADDRESS 2-46
BLOCK 2-47
DEC 2-48
DIV 2-48
HEX 2-49
MOD 2-49
MUL 2-50
SUB 2-50
QUIT 2-52
READ 2-53
RESTOREFNODE 2-54
RESTOREVOLUMELABEL 2-57
SAVE 2-59
SUBSTITUTEBYTE 2-61
SUBSTITUTEWORD 2-64
VERIFY 2-65
WRITE 2-74

CHAPTER 3 **PAGE**
BACKING UP AND RESTORING FNODES

3.1 Overview 3-1
3.2 Using FNODE Backup and Restore 3-4
3.2.1 Creating the R?SAVE FNODE Backup File 3-4
3.2.2 Backing up FNODEs on a Volume 3-5
3.2.3 Backing up the Volume Label 3-6
3.2.4 Restoring FNODEs 3-7
3.2.5 Restoring the Volume Label 3-10
3.2.6 Displaying R?SAVE FNODEs 3-11

APPENDIX A **PAGE**
STRUCTURE OF A NAMED VOLUME

A.1 Introduction A-1
A.2 Volume Structure A-1
A.3 Volume Labels A-2
A.3.1 ISO Volume Label A-3
A.3.2 iRMX® II Volume Label A-4
A.4 Initial Files A-8
A.4.1 FNODE File A-8
A.4.2 FNODE 0 (FNODE file) A-14
A.4.3 FNODE 1 (Volume Free Space Map File) A-15
A.4.4 FNODE 2 (Free FNODEs Map File) A-15
A.4.5 FNODE 3 (Accounting File) A-15
A.4.6 FNODE 4 (Bad Blocks Map File) A-16

APPENDIX A (continued) **PAGE**

A.4.7 FNODE 5 (Volume Label File) A-16
A.4.8 FNODE 6 (Root Directory) A-16
A.4.9 FNODE 7 (R?SAVE) A-17
A.4.10 Other FNODEs A-17
A.5 Long and Short Files A-17
A.5.1 Short Files A-17
A.5.2 Long Files A-19
A.6 Flexible Diskette Formats A-21

Intel® TABLES

TABLE **PAGE**

A-1 8-Inch Diskette Characteristics A-21
A-2 5 1/4-Inch Diskette Characteristics A-22

Intel® FIGURES

FIGURE **PAGE**

2-1 DISPLAYBYTE Format 2-17
2-2 LISTBADBLOCKS Format 2-44
2-3 NAMED1 Verification Output 2-67
2-4 NAMED2 Verification Output 2-68
2-5 PHYSICAL Verification Output 2-68
A-1 General Structure of Named Volumes A-2
A-2 Short File Fnode A-18
A-3 Long File FNODE A-20

CHAPTER 1
INVOKING DISKVERIFY

1.1 INTRODUCTION

When using an iRMX II application system, you will need to store data on secondary storage devices. Unfortunately, occasional power irregularities or accidental reset may destroy the index to the data on these devices, making the information inaccessible to the system. In some cases, losing even a small amount of data can render an entire volume useless.

You need a tool to examine and fix the damaged volume. This tool should enable you to determine how much of the data was damaged and help you recreate file structures on the damaged volume. The iRMX II Disk Verification Utility (DISKVERIFY) is a tool that enables you to verify the consistency and recover damaged data on iRMX volumes.

The Disk Verification Utility inspects, verifies, and corrects the data structures of iRMX named volumes. It can also verify an iRMX physical volume. The Disk Verification Utility can reconstruct the fnode file, the volume label, the file descriptor nodes (fnodes) map, the volume free space map, and the bad blocks map of the volume. In addition, with DISKVERIFY you can manipulate fnodes, bad track information, and the actual data on the volumes. The Disk Verification Utility also supports auto-volume recognition which means you can verify any iRMX named volume without detaching and attaching the device with the correct DUIB.

You can use DISKVERIFY in one of two ways:

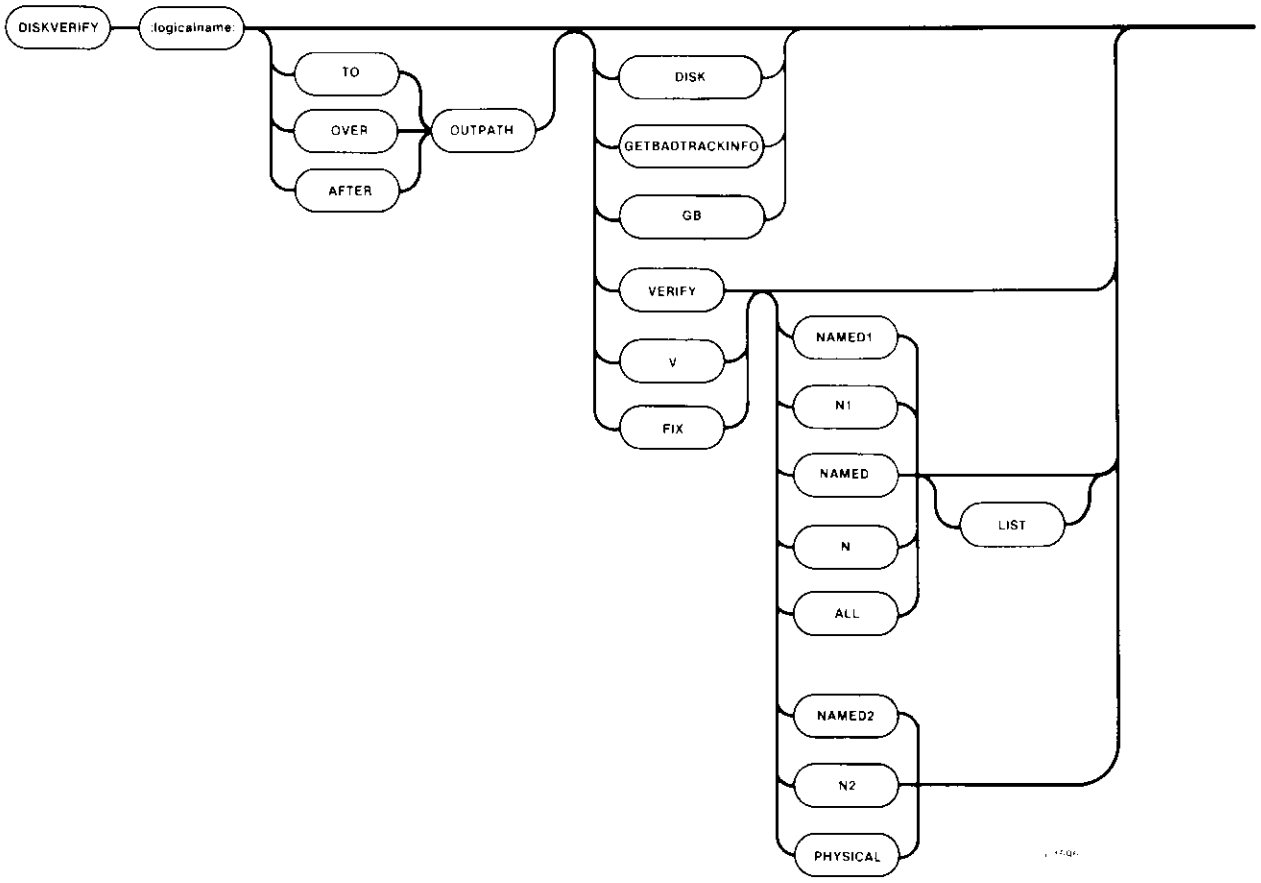
- As a single command that verifies the structures of a volume and returns control to the Human Interface
- As an interactive program that enables you to check and modify data on the volume by entering disk verification commands

To take full advantage of this utility, you must be familiar with the structure of iRMX (either iRMX I or iRMX II as the volume structure is almost the same for both) named volumes. Appendix A contains detailed information about volume structure. If you are unfamiliar with the iRMX II volume structure, you should avoid using the DISKVERIFY commands. Some commands, if not used correctly, can render your volumes unusable.

However, even if you know nothing about iRMX volume structures, you can still use the Disk Verification Utility as a single command to verify that the data structures on an iRMX volume are valid.

1.2 INVOCATION

To invoke DISKVERIFY, enter the following command:



where:

:logical name:	Logical name of the secondary storage device containing the volume to be verified.
TO	Copies the output from the Disk Verification Utility to the file specified in OUTPATH. If no "TO" is specified, output is directed to the console screen (:CO:).
OVER	Copies the output from the Disk Verification Utility over the specified file.
AFTER	Copies the output from the Disk Verification Utility beginning at the end of the specified file.

OUTPATH	Pathname of the file to receive the output from the Disk Verification Utility. If you omit this parameter and no preposition is specified, output is directed to the console screen (:CO:) by default. You cannot direct the output to a file on the volume being verified. If you attempt this, the utility returns an E$ALREADY_ATTACHED error message.

Following is a list of the DISKVERIFY options. If you invoke DISKVERIFY without specifying one of these options, you enter the interactive mode. In this case, the utility displays a header message and the utility prompt (*). You can then enter any of the DISKVERIFY commands listed in Chapter 2.

DISK	Displays the attributes of the volume being verified. If you specify this option, the utility performs the function and returns control to you at the Human Interface level. You can then enter any Human Interface command, provided that the device verified is not the system device. Any parameter after this one is ignored. Refer to the description of the DISK command in Chapter 2 for more information.
GETBADTRACKINFO or GB	Reads the bad track information from the volume and displays it. Bad track information that is redirected to a file can be used as input to the FORMAT command by removing the header information. Chapter 2 provides a complete explanation of this command.
VERIFY or V	Verifies the volume. This function and the associated options are described in detail under "VERIFY" in Chapter 2. If you specify only this option, the utility performs the NAMED verification function and returns control to you at the Human Interface level. You can then enter any Human Interface command, provided the device verified is not the system device.

FIX

Performs the same functions as VERIFY. In addition, it tries to fix several types of problems on the volume after performing the verification. You should be careful when using FIX as it changes the data on the disk (which may prove dangerous). For example, during NAMED1 verification, FIX corrects the checksums on fnodes with bad checksums. However, an fnode with a bad checksum may indicate another problem with the fnode which should not be ignored. As a result, it is recommended that you use FIX only after performing the following steps.

1. Use DISKVERIFY with the VERIFY option.
2. Examine the output and the problems on the volume to determine the type of "fix" needed.
3. If the problems can be fixed using DISKVERIFY, run DISKVERIFY with the FIX option to correct the problems.

NAMED1 or N1

VERIFY or FIX option that applies to named volumes only. This option checks the fnodes of the volume to ensure that they match the directories in terms of file type and file hierarchy. This option also checks the information in each fnode to ensure consistency.

When used with FIX, the NAMED1 option corrects bad checksums and attaches orphan fnodes to their parents. Refer to the description of the VERIFY and FIX commands in Chapter 2 for more information.

NAMED2 or N2

VERIFY or FIX option that applies to named volumes only. This option checks the allocation of fnodes and space on the volume, constructs the space and fnode bit maps to reflect the current contents of the volume, and verifies that the fnodes point to the correct locations on the volume. When used with the FIX option, NAMED2 saves the correct bit maps, that were constructed during the verification phase, on the volume. It also removes fnodes with multiple references from their illegal parents. Refer to the description of the VERIFY and FIX commands in Chapter 2 for more information.

NAMED or N

VERIFY or FIX option that performs both the NAMED1 and NAMED2 verification functions on a named volume. If you specify VERIFY or FIX with no option, the system assumes NAMED (default).

ALL	VERIFY or FIX option that applies to both named and physical volumes. For named volumes, this option performs both the NAMED and PHYSICAL verification functions. For physical volumes, this option performs only the PHYSICAL verification function.
PHYSICAL	VERIFY or FIX option that applies to both named and physical volumes. This option reads all blocks on the volume and checks for I/O errors. When used with FIX, it adds the bad blocks that it encounters to the volume's bad block map.
LIST	A control that you can use with any option that activates NAMED1 verification (NAMED, NAMED1, or ALL). When you use this option, the file information generated by VERIFY or FIX is displayed for every file on the volume, even if the file contains no errors. Refer to the description of the VERIFY and FIX commands in Chapter 2 for more information.

1.3 OUTPUT

When you enter the DISKVERIFY command, the utility responds with

```
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
```

where Vx.x is the version number of the utility. If you specify the VERIFY (or V) parameter in the DISKVERIFY command, the utility verifies the volume and displays the verification information on the screen (or copies it to the file specified by the outpath parameter). The verification information is the same as that from the VERIFY utility command. After generating the verification output, the utility returns control to the Human Interface, which prompts you for more Human Interface commands. The following is an example of such a DISKVERIFY command:

```
-DISKVERIFY :F1: VERIFY NAMED2 <CR>
iRMX II Disk Verify Utility,  Vx.x
Copyright <year> Intel Corporation

DEVICE NAME = wfd0  : DEVICE SIZE = 0003E900 : BLOCK SIZE = 0080

'NAMED2'  VERIFICATION
    BIT MAPS O.K.
-
```

If you omit the DISK or VERIFY parameter from the DISKVERIFY command, the utility does not return control to the Human Interface. Instead, it issues an asterisk (*) prompt and waits for you to enter DISKVERIFY commands. The following is an example:

```
-DISKVERIFY :F1: <CR>
iRMX II Disk Verify Utility,  Vx.x
Copyright <year> Intel Corporation
*
```

At the asterisk prompt, you can enter any of the DISKVERIFY commands listed in the DISKVERIFY COMMANDS chapter of this manual. If you enter anything else, the utility will display an error message.

NOTE

Although you can use DISKVERIFY to verify the system device (:sd:), note that all connections to this device are deleted by the operating system. After exiting, you must reboot the system or use the warm start feature (see the *Extended iRMX II System Debugger Reference Manual*).

1.4 INVOCATION ERROR MESSAGES

The following is a list of error messages you might encounter when invoking the Disk Verification Utility.

argument error	The option specified is not valid.
<logical name>, invalid logical name.	The logical name does not exist; was longer than 12 characters; contained invalid characters; or was missing a matching colon.
0045 : E$LOG_NAME_NEXIST or <logical name>, logical name does not exist	A nonexistent <logical name> was specified in either the :logical name: or outpath parameter.
<outpath> 0038 : E$ALREADY_ATTACHED	The output was directed to a file on the volume being verified.
command syntax error	A syntax error was made when entering the command.

<logical name>, outstanding connections to the device have been deleted.	This warning is not fatal and will occur every time you try to verify the system device or any other volume on which files have been attached.
<logical name> or <outpath>, invalid wildcard specification	The logical name or output pathname contained a wildcard character.
<logical name>, can't attach device	The device cannot be attached and read.
device size inconsistent size in volume label = <value1> : computed size = <value2>	When the Disk Verification Utility computed the size of the volume, the size it computed did not match the information recorded in the iRMX II volume label. The volume label may contain invalid or corrupted information. This is not a fatal error, but it is an indication that further error conditions may result during the verification session. You may have to reformat the volume or use the Disk Verification Utility to restore the volume label.
not a named disk	A NAMED, NAMED1, or NAMED2 verification was requested for a physical volume.
<partial logical name>, 0081: E$STRING_BUFFER	The logical name was longer than 14 characters in length, not including colons.
<logical name>, device does not belong to you	An attempt was made to verify a device that was attached by another user. For example, the system device is :SD: and USER is not the super user.
<logical name>, device size is zero	The logical name entered does not define a mass storage device. For example, you cannot perform DISKVERIFY on a line printer.

Intel®

CHAPTER 2
DISKVERIFY COMMANDS

2.1 INTRODUCTION

When the Disk Verification Utility issues the asterisk (*) prompt, you can enter DISKVERIFY commands to examine or change file structure information on the volume. This process usually involves reading a portion of the volume into a buffer, modifying that buffer, and writing the information back to the volume. This chapter describes the commands that enable you to perform these operations.

The commands in this chapter are presented in alphabetical order regardless of their function. The only exception is when two commands are similar, such as DISPLAYBYTE and DISPLAYWORD. In this case, the first command is explained in its alphabetical order, and the second command follows it with only the differences described.

The first occurrence of each command name is printed in blue ink and appears on the outside upper corner of the page; subsequent occurrences are printed in black ink. Blue or bolded text is also used to indicate an entry you make from your terminal.

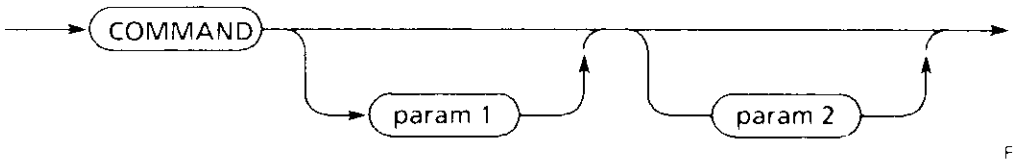
Before describing the individual commands, this chapter discusses command syntax, command names, parameters, input radices, and error messages. It also provides a command dictionary that gives a brief description of each command and the page number on which the command is found.

2.2 COMMAND SYNTAX

The syntax for each command described in this chapter is presented in a "railroad track" schematic, with syntactic elements scattered along the track. Your entrance to any given schematic is always from left to right, beginning with the command name entry.

Elements shown in uppercase characters must be typed in a command line exactly as shown in the schematic, however, you may enter them in either uppercase or lowercase. Syntactic elements shown in lowercase are generic terms, which means you must supply the specific item, such as the pathname of a file.

"Railroad sidings" go through optional parameter elements. In some cases, you have a choice of going through one of several sidings before returning to the main track. In still other cases, the main track itself diverges into two separate tracks, which means you must select one track or the other but not both. For example, a command that consists of a command name and two optional parameters would look like this:



You can enter this command in any one of these forms:

The arrows are used here to illustrate the possible flow through the tracks. They do not appear in the schematics in the rest of this chapter.

2.3 COMMAND NAMES

When you enter a DISKVERIFY command, you can enter the command name or its abbreviation (listed in this chapter), or you can enter any unique portion of the command name. For example, when specifying the DISPLAYFNODE command, you can enter any of the following:

You can also enter any other partial form of the word DISPLAYFNODE that contains at least the characters DISPLAYF.

2.4 PARAMETERS

Several DISKVERIFY commands have parameters described as being in this form:

You can also enter these parameters in this form:

For example, both of these specify a FREE command:

2.5 INPUT RADICES

DISKVERIFY always produces numerical output in hexadecimal format. You can provide input to DISKVERIFY in any one of the following three radices by including a radix character immediately after the number. The valid radix characters are

radix	character	example
hexadecimal	h or H	16h, 7CH
decimal	t or T	23t, 100T
octal	o, O, q, or Q	27o, 33Q

If you omit the radix character, DISKVERIFY assumes the number is hexadecimal.

2.6 ABORTING DISKVERIFY COMMANDS

You can abort the following DISKVERIFY commands by entering a CONTROL-C, which terminates the command and returns control to the Disk Verification Utility (not the Human Interface command level).

DISK
DISPLAYBYTE
DISPLAYDIRECTORY
DISPLAYFNODE
DISPLAYNEXTBLOCK
DISPLAYPREVIOUSBLOCK
DISPLAYWORD
EDITFNODE
EDITSAVEFNODE
FIX
GETBADTRACKINFO
LISTBADBLOCKS
SUBSTITUTEBYTE
SUBSTITUTEWORD
VERIFY

2.7 DISKVERIFY ERROR MESSAGES

Each DISKVERIFY command can generate a number of error messages, which indicate errors in the way the command was specified or problems with the volume itself. The following messages can be generated by many of the commands (each command description lists the error messages generated by the particular command):

block I/O error	The utility attempted to read or write a block on the volume and found that the block was physically damaged and therefore, could not complete the requested command. Or, an attempt was made to write a block to a disk volume that is write protected. The error message states whether read or write was performed and the number of the block causing the error.
command syntax error	A syntax error was made in a command.
illegal command	The command specified is not a valid DISKVERIFY command.

fnode file/space map file inconsistent	One of the files, R?SAVE or R$FNODEMAP, is damaged and DISKVERIFY cannot perform further verification.
argument error	The command was missing an argument, or the argument was illegally specified.
not a named disk	The device is not a named volume (a tape, for example) or the iRMX volume label, obtained when DISKVERIFY begins processing, contains invalid information. If the label contains invalid information, the utility (in some cases) can assume that a named volume is a physical volume. In this case, the commands that apply to named volumes only (such as DISPLAYFNODE, DISPLAYDIRECTORY, and VERIFY NAMED) issue this message. If you are sure the volume is a named volume, this message may indicate that the iRMX II volume label is corrupted. (If the file was formatted with the RESERVE option of the FORMAT command, DISKVERIFY issues this message only if both volume labels are corrupted. When only the volume label is invalid, the duplicate in the save area is used.)
seek error	The utility unsuccessfully attempted to seek to a location on the volume. This error normally results from invalid information in the iRMX II volume label or in the fnodes. Or, a new volume was inserted after DISKVERIFY was invoked.

2.8 COMMAND DICTIONARY

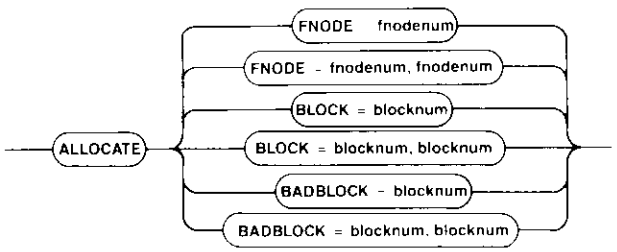
The command dictionary below lists the DISKVERIFY commands in alphabetical order and provides a brief functional description of each command. Following each command name is its unique abbreviation, if any. For quick reference, you can locate the command using the page headers remaining in this chapter.

Command	Synopsis
ALLOCATE	Marks a particular fnode or volume block as allocated
BACKUPFNODES BF	Copies current fnode file into a backup file named R?SAVE

Command	Synopsis
DISK	Displays the attributes of the volume being verified.
DISPLAYBYTE DB or D	Displays the working buffer in byte format
DISPLAYWORD DW	Displays the working buffer in word format
DISPLAYDIRECTORY DD	Displays directory contents
DISPLAYFNODE DF	Displays the specified fnode information
DISPLAYSAVEFNODE DSF	Displays the fields of a single fnode in the R?SAVE file
DISPLAYNEXTBLOCK DNB or > or <CR>	Displays the "next" volume block
DISPLAYPREVIOUSBLOCK DPB or <	Displays the "previous" volume block
EDITFNODE EF	Edits the specified fnode
EDITSAVEFNODE ESF	Edits the specified saved fnode
EXIT E	Exits the Disk Verification Utility
FIX	Verifies the disk and fixes inconsistencies
FREE	Marks a particular fnode or volume block as free
GETBADTRACKINFO GB	Displays the bad track information

Command	Synopsis
HELP H	Lists the DISKVERIFY commands
LISTBADBLOCKS LBB	Displays all the bad blocks on the volume
Miscellaneous Commands	Perform useful arithmetic and conversion functions; the commands include ADD, SUB, MUL, DIV, MOD, HEX, DEC, ADDRESS,and BLOCK
QUIT Q	Exits the Disk Verification Utility
READ R	Reads a volume block into the working buffer
RESTOREFNODE RF	Copies one fnode (or range of fnodes) from the R?SAVE file to the fnode file
RESTOREVOLUMELABEL RVL	Copies the duplicate volume label to the volume label offset on track 0
SAVE	Writes the updated fnode map, free space map, and bad block map to the volume
SUBSTITUTEBYTE SB or S	Modifies the contents of the working buffer in byte format
SUBSTITUTEWORD SW	Modifies the contents of the working buffer in word format
VERIFY V	Verifies the volume
WRITE W	Writes the working buffer to the volume

This command designates file descriptor nodes (fnodes) and volume blocks as allocated. You can also use this command to designate one or a range of volume blocks as "bad." The format of the ALLOCATE command is as follows:



INPUT PARAMETERS

fnodenum	Number of the fnode to allocate. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. Two fnode values separated by a comma signifies a range of fnodes.
blocknum	Number of the volume block to allocate. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume. Two block numbers separated by a comma signifies a range of block numbers.

OUTPUT

If you are using ALLOCATE to allocate fnodes, ALLOCATE displays the following message:

```
<fnodenum>, fnode marked allocated
```

where <fnodenum> is the number of the fnode that the utility designated as allocated.

If you are using ALLOCATE to allocate volume blocks, ALLOCATE displays the following message:

```
<blocknum>, block marked allocated
```

where <blocknum> is the number of the volume block that the utility designated as allocated.

If you are using ALLOCATE to designate one or more volume blocks as "bad," ALLOCATE displays the following message:

```
<blocknum>, block marked bad
```

where <blocknum> is the number of the volume block that the utility designated as "bad." If this block was not allocated before you attempt to designate it as "bad," ALLOCATE also displays

```
<blocknum>, block marked allocated
```

ALLOCATE checks the allocation status of fnodes or blocks before allocating them. Therefore, if you specify ALLOCATE for a block or fnode already allocated, ALLOCATE returns one of the following messages:

```
<fnodenum>, fnode already marked allocated
```

```
<blocknum>, block already marked allocated
```

```
<blocknum>, block already marked bad
```

DESCRIPTION

Fnodes are data structures on the volume that describe the files on the volume. They are created when the volume is formatted. An allocated fnode is one that represents an actual file. ALLOCATE designates fnodes as allocated by updating the FLAGS field of the fnode and free-fnodes-map file with this information.

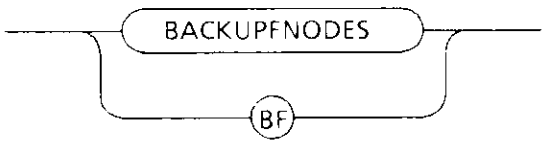
An allocated volume block is a block of data storage that is part of a file; it is not available to be assigned to a new file. ALLOCATE designates volume blocks as allocated by updating the volume free-space-map with this information.

When you use ALLOCATE to designate bad blocks, it not only updates the volume free-space-map but also marks an associated bit as "bad" in the bad blocks file.

ERROR MESSAGES

argument error	A syntax error was made in the command or a nonnumeric character was specified in the blocknum or fnodenum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.
<fnodenum>, fnode out of range	The fnode number specified was larger than the largest fnode number in the volume.
no badblocks file	The volume does not have a bad blocks file. This message could appear if an earlier version of the Human Interface FORMAT command was used when the disk was formatted.

This command copies the current fnode file into a designated fnode backup file named R?SAVE. R?SAVE must have been reserved when the volume was formatted. (That is, the RESERVE option of the FORMAT command must have been specified.) The format of the BACKUPFNODES command is as follows:



F 02'5

INPUT PARAMETERS

None.

OUTPUT

BACKUPFNODES displays the following message:

```
fnode file backed up to save area
```

DESCRIPTION

The BACKUPFNODES command ensures against data loss that occurs when the fnode file is damaged or destroyed. To use this command, you must have formatted the volume using the FORMAT command (V1.1 or later) to create a special reserve area (R?SAVE). A switch in the FORMAT command (the RESERVE switch) controls the creation of R?SAVE. If you did not specify the RESERVE parameter when the volume was formatted, the BACKUPFNODES command will be unable to copy the fnode file to R?SAVE. An error message will be returned indicating that no save area has been reserved. In this case, the volume must be reformatted if you wish to use the BACKUPFNODES command.

The FORMAT command writes the initialized copy of the fnode file into R?SAVE. Therefore, you do not have to use BACKUPFNODES to back up a newly formatted volume. Subsequently, you can routinely (for example, once a day) backup fnodes to assure that the data in R?SAVE matches the data in the fnode file. You can do this by using either the BACKUPFNODES command or the Human Interface SHUTDOWN command with the BACKUP option. (For more information on SHUTDOWN, see the *Operator's Guide to the Extended iRMX II Human Interface*.)

BACKUPFNODES

NOTE

Be sure that the current fnode file is valid before executing the BACKUPFNODE command (using NAMED verification).

ERROR MESSAGES

argument error	When the command was entered, an argument was supplied. BACKUPFNODES does not accept an argument.
no save area was reserved when volume was formatted	The volume has not been formatted to support fnode backup. To allow future use of backupfnodes on this volume, you should invoke the Human Interface BACKUP command to save the data on the volume, reformat the volume with a save area (using the RESERVE option of the FORMAT command), and finally, restore the volume data.
not a named disk	The volume specified when the Disk Verification Utility was invoked is a physical volume, not a named volume.

EXAMPLE

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
*verify NAMED <CR>
            .
            .
            .
   BIT MAPS O.K.
*backupfnodes <CR>   or bf <CR>
fnode file backed up to save area
*
```

This command displays the attributes of the volume being verified. You can abort this command by typing a CONTROL-C. The format of the DISK command is as follows:

x225

INPUT

None.

OUTPUT

The output of the DISK command depends on whether the volume is formatted as a physical or named volume. For a physical volume, the DISK command displays the following information:

```
          device name  = <devname>
       physical disk
device granularity  = <devgran>
        block size  = <devgran>
  number of blocks  = <numblocks>
       volume size  = <size>
```

where:

<devname>	Name of the device containing the volume. This is the physical name of the device, as specified in the ATTACHDEVICE Human Interface command.
<devgran>	Granularity of the device, as defined in the Device Unit Information Block (DUIB) for the device. Refer to the *Guide to the Extended iRMX II Interactive Configuration Utility* for more information about DUIBs. For physical devices, this is also the volume block size.
<numblocks>	Number of volume blocks in the volume.
<size>	Size of the volume, in bytes.

For a named volume, the DISK command displays the following information:

```
             device name = <devname>
  named disk, volume name = <volname>
       device granularity = <devgran>
                block size = <volgran>
          number of blocks = <numblocks>
    number of free blocks = <numfreeblocks>
             volume size = <size>
              interleave = <inleave>
           extension size = <xsize>
         number of fnodes = <numfnodes>
    number of free fnodes = <numfreefnodes>
       save area reserved = (yes/no)
```

The <devname>, <devgran>, <numblocks>, and <size> fields are the same as for physical files. The remaining fields are as follows:

<volname>	Name of the volume, as specified when the volume was formatted.
<volgran>	Volume granularity, as specified when the volume was formatted.
<numfreeblocks>	Number of available volume blocks in the volume.
<inleave>	The interleave factor for a named volume.
<xsize>	Size, in bytes, of the extension data portion of each file descriptor node (fnode).
<numfnodes>	Number of fnodes in the volume. The fnodes were created when the volume was formatted.
<numfreefnodes>	Number of available fnodes in the named volume.
save area reserved	Indicates whether the R?SAVE file is reserved for volume label and fnode file backups.

Refer to Appendix A of this manual or to the description of the FORMAT command in the *Operator's Guide to the Extended iRMX II Human Interface* for more information about the named disk fields.

DESCRIPTION

The DISK command displays the attributes of the volume. The format of the output from DISK depends on whether the volume is formatted as a named or physical volume.

ERROR MESSAGES

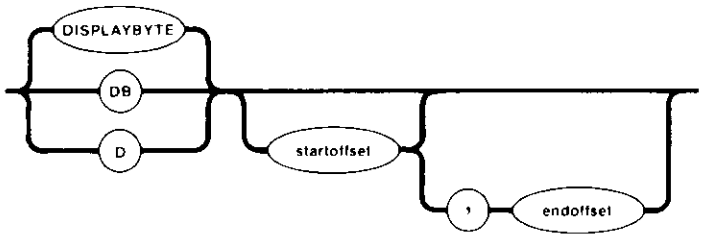
None.

EXAMPLE

The following example shows the output of the DISK command for an 5.25-inch diskette.

```
super- diskverify :f0: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation

*disk  <CR>

                    device name = wmfdx0
          named disk, volume name = rmx286
               device granularity = 0200
                      block size = 0200
                number of blocks = 0000027C
          number of free blocks = 000001E9
                    volume size = 0004F800
                     interleave = 0005
                  extension size = 03
                number of fnodes = 00CF
          number of free fnodes = 00BE
             save area reserved = no
```

This command displays the specified portion of the working buffer in BYTE format. It displays the buffer in 16-byte rows. You can abort this command by typing a CONTROL-C. The format of the DISPLAYBYTE command is as follows:



x 226

INPUT PARAMETERS

startoffset	Number of the byte, relative to the start of the buffer, that begins the display. DISPLAYBYTE starts the display with the row containing the specified offset. If you omit this parameter and the endoffset parameter, DISPLAYBYTE displays the entire working buffer.
endoffset	Number of the byte, relative to the start of the buffer, that ends the display. If you omit this parameter, DISPLAYBYTE displays only the row indicated by startoffset. However, if you omit both startoffset and endoffset, DISPLAYBYTE displays the entire working buffer.

OUTPUT

In response to the command, DISPLAYBYTE displays the specified portion of the working buffer in rows, with 16 bytes displayed in each row. Figure 2-1 illustrates the format of the display.

As Figure 2-1 shows, DISPLAYBYTE begins by listing the block number where data resides in the working buffer. It then lists the specified portion of the buffer, providing the column numbers as a header and beginning each row with the relative address of the first byte in the row. It also includes, at the right of the listing, the ASCII equivalents of the bytes, if the ASCII equivalents are printable characters. (If a byte is not a printable character, DISPLAYBYTE displays a period in the corresponding position.)

```
*displaybyte 7,13 <CR>

BLOCK NUMBER = blocknum

offset  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII STRING
 0000  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   ............
 0010  61 6E 20 65 F8 61 6D 70 6C 65 20 20 20 20 20 20   an example
```

Figure 2-1. DISPLAYBYTE Format

DESCRIPTION

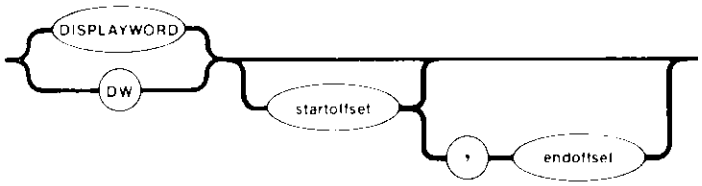
DISKVERIFY maintains a working buffer for READ and WRITE commands. The size of the buffer is equal to the volume's granularity value. After you read a volume block of memory into the working buffer with the READ command, you can display part or all of that buffer, in BYTE format, by entering the DISPLAYBYTE command. DISPLAYBYTE displays the hexadecimal value for each byte in the specified portion of the buffer.

If you omit all parameters, DISPLAYBYTE displays the entire block stored in the working buffer.

ERROR MESSAGES

argument error	A syntax error was made in the command or a nonnumeric character was specified in one of the offset parameters.
<offset>, invalid offset	Either a larger value was specified for startoffset than for endoffset or an offset value larger than the number of bytes in the block was specified.

This command is the same as the DISPLAYBYTE command, except that it displays the working buffer in WORD format, 8-words per row. The format of the DISPLAYWORD command is as follows:



EXAMPLES

Assuming that the volume granularity is 128 bytes and that you have read block 20H into the working buffer with the READ command, the following command displays that block in WORD format.

```
*DISPLAYWORD   <CR>

BLOCK NUMBER = 20

offset     0     2     4     6     8     A     C     E
 0000    0000  0000  0000  0000  0000  0000  0000  0000
 0010    0000  0080  0000  0000  0000  0001  FF0F  00FF
 0020    0000  0000  0500  0000  0000  0025  0108  FFFF
 0030    1F25  0000  002E  0000  1F25  0000  002B  0000
 0040    0001  0000  0001  0080  0000  0000  0000  0000
 0050    0000  0000  0000  0000  0000  0000  0000  0000
 0060    0000  0000  0000  0000  0000  0000  0080  0000
 0070    0000  0000  0001  FF0F  00FF  0000  0000  0500
*
```

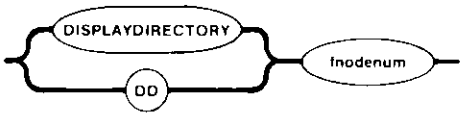
The following command displays the portion of the block that contains the offsets 31h through 45h (words beginning at odd addresses).

```
*DW 31, 45  <CR>
BLOCK NUMBER = 20

offset     0     2     4     6     8     A     C     E
 0031    001F  2E00  0000  2500  001F  2B00  0000  0100
 0041    0000  0100  8000  0000  0000  0000  0000  0000
*
```

The following command displays the portion of the block that contains the offsets 30h through 45h (words beginning at even addresses).

```
*DISPLAYWORD 30, 45 <CR>
BLOCK NUMBER = 20

offset     0     2     4     6     8     A     C     E
 0030    1F25  0000  002E  0000  1F25  0000  002B  0000
 0040    0001  0000  0001  0080  0000  0000  0000  0000
*
```

This command lists all the files contained in a directory. You can abort this command by typing a CONTROL-C. The format of the DISPLAYDIRECTORY command is as follows:



x 227

INPUT PARAMETER

fnodenum	Number of the fnode that corresponds to a directory file. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. DISPLAYDIRECTORY lists all files or directories contained in this directory.

OUTPUT

In response to the command, DISPLAYDIRECTORY lists information about all files contained in the specified directory. The format of this display is as follows:

```
FILE NAME  FNODE   TYPE    FILE NAME FNODE   TYPE    FILE NAME FNODE   TYPE

<filenam> <fnode> <type>  <filenam> <fnode> <type>  <filenam> <fnode> <type>
<filenam> <fnode> <type>  <filenam> <fnode> <type>  <filenam> <fnode> <type>
          .                         .                         .
          .                         .                         .
          .                         .                         .
```

where:

<filenam>	Name of the file or directory contained in the directory.
<fnode>	Number of the fnode that describes the file.

type> Type of the file. The <type> can be

Type of file	Description
DATA	data files
DIR	directory files
SMAP	volume free space map
FMAP	free fnodes map
BMAP	bad blocks map
VLAB	volume label file

DESCRIPTION

DISPLAYDIRECTORY displays a list of files contained in the specified directory, along with their fnode numbers and types. You can then use other DISKVERIFY commands to examine the individual files.

ERROR MESSAGES

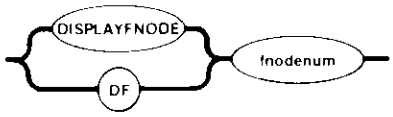
argument error	A nonnumeric character was specified in the fnodenum parameter.
<fnodenum>, fnode not allocated	The number specified for the fnodenum parameter does not correspond to an allocated fnode. This fnode does not represent an actual file.
<fnodenum>, not a directory fnode	The number specified for the fnodenum parameter is not an fnode for a directory file.
<fnodenum>, fnode out of range	The number specified for the fnodenum parameter is larger than the largest fnode number on the volume.

EXAMPLE

The following command lists the files contained in the directory whose fnode is fnode 6.

```
*DISPLAYDIRECTORY 6  <CR>

     FILE NAME  FNODE TYPE  FILE NAME  FNODE TYPE     FILE NAME  FNODE TYPE
    R?SPACEMAP   0001 SMAP R?FNODEMAP   0002 FMAP R?BADBLOCKMAP   0004 BMAP
 R?VOLUMELABEL   0005 VLAB      R?SAVE   0007 DATA         RMX286   0008 DIR
        MYFILE   0009 DATA    YOURFILE   000A DATA       ONEFILE   000B DATA
*
```

This command displays the fields associated with an fnode. You can abort this command by typing a CONTROL-C. The format of the DISPLAYFNODE command is as follows:



x-228

INPUT PARAMETER

fnodenum	Number of the fnode to be displayed. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted.

OUTPUT

In response to this command, DISPLAYFNODE displays the fields of the specified fnode. The format of the display is as follows:

```
Fnode number = <fnodenum>
path name: <pathname>
                  flags : <flgs>
                   type : <typ>
      file gran/vol gran : <gran>
                  owner : <own>
 create,access,mod times : <crtime>, <acctime>, <modtime>
   total size,total blks : <totsize>, <totblks>
       block pointer (1) : <blks>, <blkptr>
       block pointer (2) : <blks>, <blkptr>
       block pointer (3) : <blks>, <blkptr>
       block pointer (4) : <blks>, <blkptr>
       block pointer (5) : <blks>, <blkptr>
       block pointer (6) : <blks>, <blkptr>
       block pointer (7) : <blks>, <blkptr>
       block pointer (8) : <blks>, <blkptr>
              this size : <thissize>
               id count : <count>
            accessor (1) : <access>, <id>
            accessor (2) : <access>, <id>
            accessor (3) : <access>, <id>
       parent, checksum  : <prnt>, <checksum>
                 aux(*) : <auxbytes>
```

where:

<fnodenum> Number of the fnode being displayed. If the fnode does not describe an actual file (that is, if it is not allocated), the following message appears next to this field:

*** ALLOCATION STATUS BIT IN THIS FNODE NOT SET ***

In this case, the fnode fields are normally set to zero.

<pathname> Full pathname of the file described by the fnode. This field is not displayed if the fnode does not describe a file.

<flgs> A word defining the attributes of the file. Significant bits of this word are as follows:

Bit	Meaning
0	Allocation status. This bit is set to 1 for allocated fnodes and 0 for free fnodes.
1	Long or short file attribute. This bit is set to 1 for long files and 0 for short files.
5	Modification attribute. This bit is set to 1 whenever a file is modified.
6	Deletion attribute. This bit is set to 1 to indicate a temporary file or a file to be deleted.

The DISPLAYFNODE command displays a message next to this field to indicate whether the file is a long or short file.

<typ> Type of file. This field contains a value and a description which is displayed next to the value. The possible values and descriptions are as follows:

Value	Descriptions
00	fnode file
01	volume map file
02	fnode map file
03	account file
04	bad block file
06	directory file
08	data file
09	volume label file

<gran> File granularity, specified as a multiple of the volume granularity.

<own> User ID of the owner of the file.

<crtime> <acctime> <modtime>	Time and date of file creation, last access, and last modification. These values are expressed as the time, in seconds, since January 1, 1978.
<totsize>	Total size, in bytes, of the actual data in the file.
<totblks>	Total number of volume blocks used by the file, including indirect block overhead.
<blks>, <blkptr>	Values that identify the data blocks of the file. For short files, each <blks> parameter indicates the number of volume blocks in the data block, and each <blkptr> is the number of the first such volume block. For long files, each <blks> parameter indicates the number of volume blocks pointed to by an indirect block, and each <blkptr> is the block number of the indirect block.
<thissize>	Size in bytes of the total data space allocated to the file, minus any space used for indirect blocks.
<count>	Number of user IDs associated with the file.
<access>, <id>	Each pair of fields indicates the access rights for the file and the ID of the user who has that access ID. Bits in the <access> field are set to indicate the following access rights:

Bit	Data File Operation	Directory Operation
0	delete	delete
1	read	list
2	append	add entry
3	update	change entry

The first ID listed is the owner's ID.

<prnt>	Fnode number of the directory file that contains the file.
<checksum>	Checksum of the fnode.
<auxbytes>	Auxiliary bytes associated with the file.

Appendix A contains a more detailed description of the fnode fields.

DESCRIPTION

Fnodes are system data structures on the volume that describe the files on the volume. The fnode structures are created when the volume is formatted. Each time a file is created on the volume, the iRMX II Basic I/O System allocates an fnode for the file and fills in the fnode fields to describe the file. The DISPLAYFNODE command enables you to examine these fnodes and determine where the data for each file resides.

DISPLAYFNODE

ERROR MESSAGES

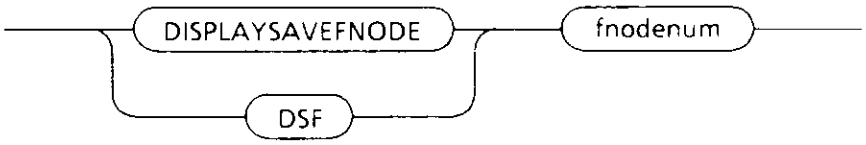
argument error	The value entered for the fnodenum parameter was not a legitimate fnode number.
<fnodenum>,fnode out of range	The number specified for the fnodenum parameter is larger than the largest fnode number on the volume.
Unable to get pathname <reason>	The pathname specified could not be retrieved. Possible causes of this error are seek error, I/O error, or invalid parent.

EXAMPLE

The following example displays fnode 10 of a volume. This fnode represents a directory.

```
*DISPLAYFNODE 10   <CR>

Fnode number = 10
path name : /MYDIR
                   flags : 0025 =>short file
                    type : 06 =>directory file
      file gran/vol gran : 01
                   owner : FFFF
 create,access,mod times : 10219017, 10219E58, 10219E58
  total size,total blocks: 00000360, 00000001
       block pointer (1) : 0001, 000050
       block pointer (2) : 0000, 000000
       block pointer (3) : 0000, 000000
       block pointer (4) : 0000, 000000
       block pointer (5) : 0000, 000000
       block pointer (6) : 0000, 000000
       block pointer (7) : 0000, 000000
       block pointer (8) : 0000, 000000
              this size  : 00000400
               id count  : 0001
            accessor (1) : 0F, FFFF
            accessor (2) : 00, 0000
            accessor (3) : 00, 0000
       parent, checksum  : 0006, 0000
                 aux(*)  : 000000
*
```

This command is identical to DISPLAYFNODE, except the DISPLAYSAVEFNODE takes the fnode information from the R?SAVE file, and displays the fnode as saved. R?SAVE must have been reserved when the volume was formatted. (That is, the RESERVE option in the FORMAT command must have been specified.) The format of the DISPLAYSAVEFNODE command is as follows:

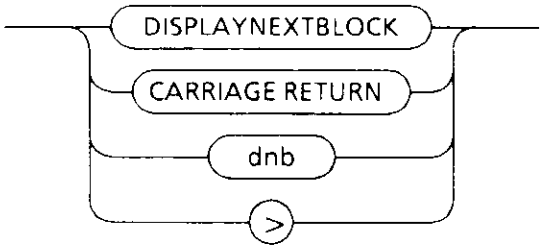


F 0212

ERROR MESSAGES

argument error	When the command was entered, no argument was supplied. DISPLAYSAVEFNODE requires a designation of the fnode number.
<fnodenum>, fnode out of range	The number specified for the fnodenum parameter is larger than the largest fnode number on the volume.
no save area was reserved when volume was formatted	The volume was not formatted to support backup fnodes. This means the RESERVE option was not specified when the volume was formatted.
Unable to get pathname <reason>	The pathname specified could not be retrieved. Possible causes of this error are seek error, I/O error, or invalid parent.

This command displays the "next" volume block. (The "next" volume block is the block immediately following the block currently in the working buffer.) The display format can be either WORD or BYTE. The utility remembers the mode in which you displayed the volume block currently in the working buffer, and it displays the next block in that format. So, if you used DISPLAYBYTE to display the current volume block, the next volume block appears in BYTE format; if you used DISPLAYWORD, the next volume block appears in WORD format. DISPLAYNEXTBLOCK uses the BYTE format as a default if you have not yet displayed a volume block. You can abort this command by typing a CONTROL-C. The format of the DISPLAYNEXTBLOCK command is as follows:



F-0205

OUTPUT

In response to the command, DISPLAYNEXTBLOCK reads the "next" volume block into the working buffer and displays it on the screen.

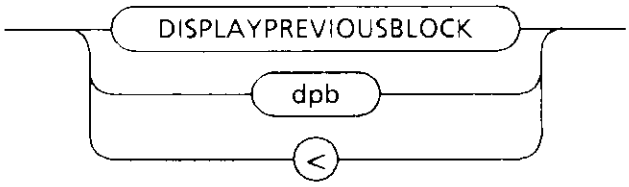
DESCRIPTION

The DISPLAYNEXTBLOCK command copies the "next" volume block from the volume to the working buffer and displays it at your terminal. It destroys any data currently in the working buffer. Once the block is in the working buffer, you can use SUBSTITUTEBYTE and SUBSTITUTEWORD to change the data in the block. Finally, you can use the WRITE command to write the modified block back out to the volume.

NOTE

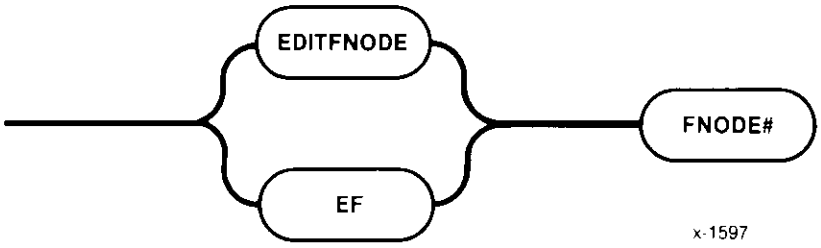
If you specify the DISPLAYNEXTBLOCK command at the end of the volume, the utility "wraps around" and displays the first block in the volume.

This command is identical to DISPLAYNEXTBLOCK, except that it displays the volume block preceding the current block in the working buffer. The format of the DISPLAYPREVIOUSBLOCK command is as follows:



F 0206

This command allows you to edit values within a specified fnode. It can be aborted by entering CONTROL-C. The format of the EDITFNODE command is



INPUT PARAMETER

fnode — Number of the fnode to edit. This number can be in the range of 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted.

OUTPUT

When EDITFNODE is invoked it displays the following message:

```
Fnode number = nnnn
```

where nnnn is the number of the fnode you want to edit. The first field of the fnode is displayed with its current value, as follows:

```
flags(xxxx):
```

where xxxx is the current value of the flags field. From this point on, you can edit the fnode fields, one at a time. After the last fnode field has been edited or a "Q" has been entered while in edit mode, the following query appears on the screen and the modified fnode is displayed.

```
Write back?
```

A response of "Yes" causes the fnode with the modified values to be written on the volume and the following message to be displayed:

```
Fnode has been updated
```

Any other response causes the fnode to remain unchanged and the following message to be displayed:

```
Fnode not changed
```

DESCRIPTION

EDITFNODE is used to change values within a specified fnode. When it is invoked, it displays the message shown above. Once you receive the invocation message, you can edit the fnode, one field at a time. The first field, flags, is displayed upon invocation (as shown above). The current value of each field is displayed followed by a colon. EDITFNODE then waits for one of the following responses from the terminal.

Response	Meaning
<CR>	No modification to the field.
numerical value <CR>	The new value to be assigned. This value is always interpreted as hexadecimal.
Quit or Q or q <CR>	Skip the remaining fields and display the query.

Any response, other than those listed above, causes the field to remain unchanged, and the next field to be displayed.

Once the fnode has been updated, you can use DISPLAYFNODE to examine the contents of the fnode and the changes you made. Changing the contents of an fnode causes it to have a bad checksum. Use FIX with the NAMED1 option to correct it. For more details, see the explanation of FIX later in this chapter.

ERROR MESSAGES

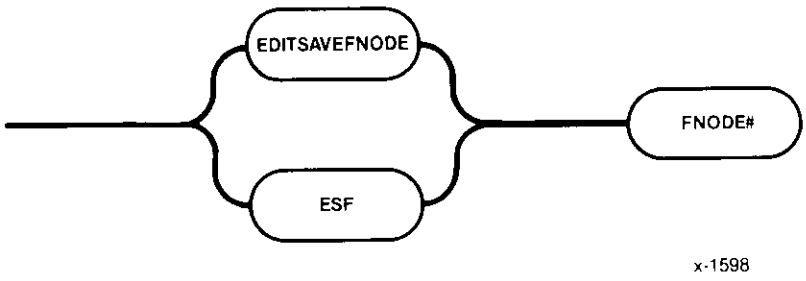
argument error	The option specified is not valid.
<fnode num>, fnode out of range	The fnode number specified was larger than the largest fnode number on the volume.
Error in Input	Invalid input was entered while editing an entry.

EXAMPLE

The following example illustrates using EDITFNODE to edit fnode 10.

```
*editfnode 10 <CR>
fnode number = 10
flags(0025):<CR>
type(0006):<CR>
file gran/vol gran(01): <CR>
owner(0FFFF):  0 <CR>
create time(10219CB2): q <CR>

Entering "q" causes the modified fnode to be displayed.

                  flags : 0025 =>short file
                   type : 06 =>directory file
      file gran/vol gran : 01
                  owner : 0000
 create,access,mod times : 10219CB2, 10219CC8, 10219CC8
  total size,total blocks: 00000360, 00000001
       block pointer (1) : 0001, 000050
       block pointer (2) : 0000, 000000
       block pointer (3) : 0000, 000000
       block pointer (4) : 0000, 000000
       block pointer (5) : 0000, 000000
       block pointer (6) : 0000, 000000
       block pointer (7) : 0000, 000000
       block pointer (8) : 0000, 000000
              this size : 00000400
               id count : 0001
           accessor (1) : 0F, FFFF
           accessor (2) : 00, 0000
           accessor (3) : 00, 0000
       parent, checksum  : 0006, 0000
                aux(*) : 000000
Write back? yes  <CR>
Fnode has been updated
*
```

EDITSAVEFNODE is identical to EDITFNODE, except that it allows you to edit an fnode from the R?SAVE file. (R?SAVE must have been reserved when the volume was formatted.) In addition, it designates the fnode as saved when displaying the fnode number. You can abort this command by entering CONTROL-C. The format of the EDITSAVEFNODE command is

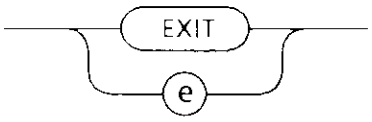


ERROR MESSAGES

The error messages are the same as in EDITFNODE with the addition of the following message.

no save area was reserved when volume was formatted	The volume was not formatted to support backup fnodes. This means the RESERVE option was not specified when the volume was formatted.

This command exits the Disk Verification Utility and returns control to the Human Interface command level. The format of the EXIT command is as follows:



F-0207

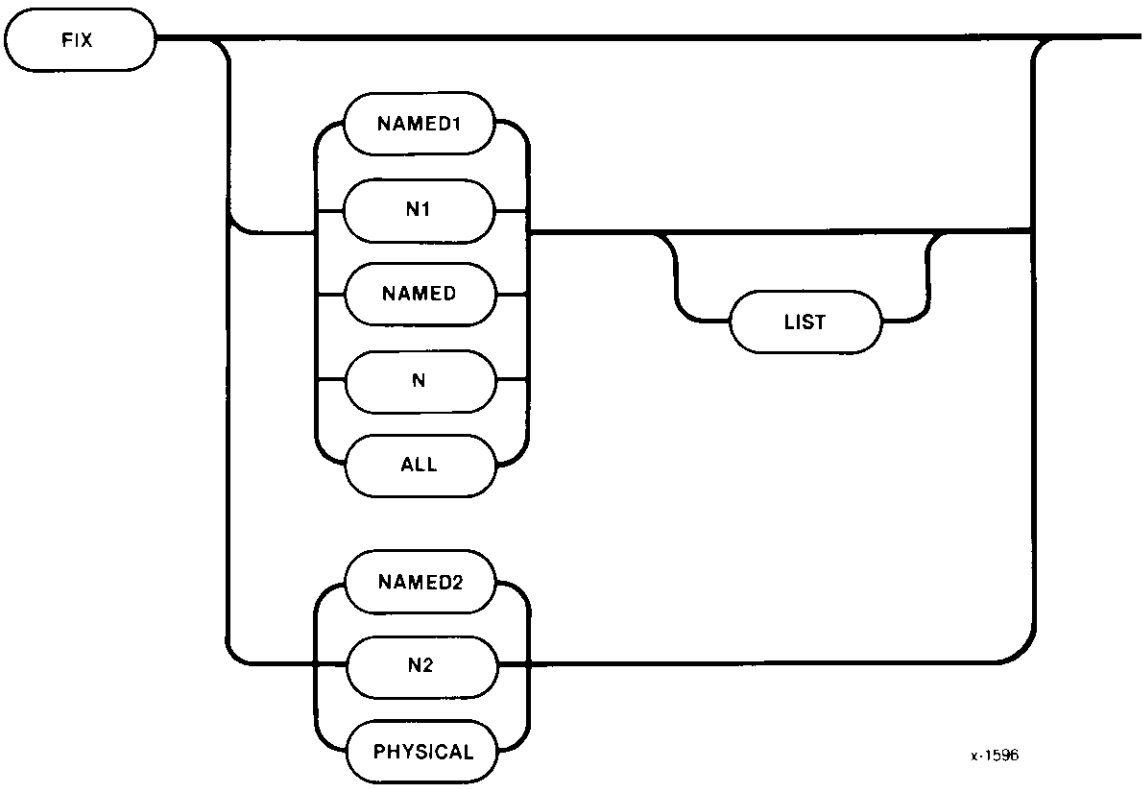
This command is identical to the QUIT command.

NOTE

Although you can use DISKVERIFY to verify the system device (:sd:), note that all connections to this device are deleted by the operating system. After exiting, you must reboot the system or use the warm start feature (see the *Extended iRMX II System Debugger Reference Manual*).

This command verifies the volume in the same way as the VERIFY command to determine if the data on the volume is consistent. In addition, this command "fixes" various kinds of inconsistencies discovered during verification. You can abort this command by entering CONTROL-C. (CONTROL-C is ignored when FIX is writing to the volume in order to prevent inconsistencies on the volume.)

Because FIX and VERIFY perform the same verification functions and generate the same error messages, the command description given below describes only the additional functions of FIX. For a complete explanation of the verify functions, see the VERIFY command described later in this chapter. The format of the FIX command is:



INPUT PARAMETERS

NAMED1 or N1	Performs NAMED1 verification and fixes the following inconsistencies: • Fixes bad checksums • Attaches orphan fnodes to their parents. An orphan fnode is an fnode contained within a directory and whose parent field does not point back to this directory. If the parent field of the specified fnode points to a second valid directory, and the second directory also points to the fnode, no fix is performed since the specified fnode belongs to an existing directory. This is a case of multiple references (discussed in NAMED2). If the parent field does not point to a valid parent, the parent field is fixed to point to the directory that contains this fnode in its file list.
NAMED2 or N2	Performs NAMED2 verification and fixes the following inconsistencies: • Removes fnodes from their illegal parents. If there is a multiple reference to an fnode, the fnode is removed from the directories that it does not point to (if FIX was performed with NAMED1, the fnode should now point to one valid parent). • Saves fnode and block bit maps on completion of NAMED2.
NAMED or N	Performs both the NAMED1 and NAMED2 verification functions on a named volume and fixes the inconsistencies defined for these options.
ALL	Performs all operations appropriate to the volume. For named volumes, this option performs both the NAMED and PHYSICAL verification functions. For physical volumes, this option performs only the PHYSICAL verification function. For both NAMED and PHYSICAL volumes, ALL performs the fixes for the relevant verifications.
PHYSICAL	Performs PHYSICAL verification and saves the bad block bit map.
LIST	Lists the file information displayed in Figure 2-3 for any verification that includes NAMED1.

OUTPUT

FIX produces the same output as the VERIFY command (see Figures 2-3, 2-4, and 2-5) with additional messages displayed when an inconsistency is fixed. NAMED1 output includes these messages.

```
Checksum Fixed
fnode nnnn was attached to parent nnnn
```

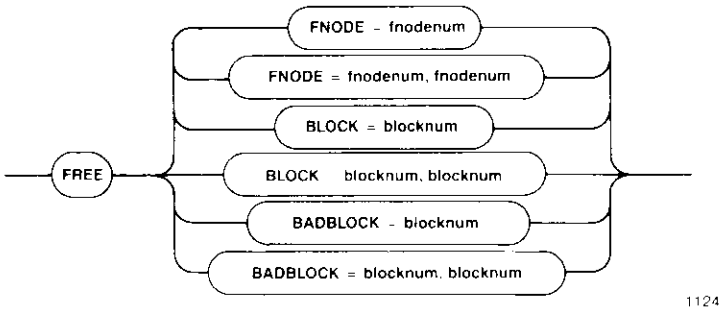
The first message appears after a bad checksum is fixed. The second message is displayed when the parent field of an fnode is modified to point to a valid parent.

NAMED2 displays this message when an fnode with multiple references is removed from the directory.

```
fnode removed from this directory
```

If an fnode exists on a disk and is marked allocated, but has not been referenced, FIX issues a warning message and asks if you want to save the bit maps. This prevents SAVE from freeing this fnode and its blocks and possibly causing a file to be lost.

This command designates fnodes and volume blocks as free (unallocated). It also removes volume blocks from the bad blocks file. The format of the FREE command is as follows:



INPUT PARAMETERS

fnodenum	Number of the fnode to free. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. Two fnode values separated by a comma signify a range of fnodes.
blocknum	Number of the volume block to free. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume. Two block numbers separated by a comma signify a range of block numbers.

OUTPUT

If you are using FREE to deallocate fnodes, FREE displays the following message:

```
<fnodenum>, fnode marked free
```

where <fnodenum> is the number of the fnode that the utility designated as free.

If you are using FREE to deallocate volume blocks, FREE displays the following message:

```
<blocknum>, block marked free
```

where <blocknum> is the number of the volume block that the utility designated as free.

If you are using FREE to designate one or more "bad" blocks as "good," FREE displays the following message:

```
<blocknum>, block marked good
```

where <blocknum> is the number of the volume block that the utility designated as "good."

FREE checks the allocation status of fnodes or blocks before freeing them. Therefore, if you specify FREE for a block or fnode that is already unallocated, FREE returns one of the following messages:

```
<fnodenum>, fnode already marked free
```

```
<blocknum>, block already marked free
```

```
<blocknum>, block already marked good
```

DESCRIPTION

Free fnodes are fnodes for which no actual files exist. FREE designates fnodes as free by updating both the FLAGS field of the fnode and the free fnodes map file.

Free volume blocks are blocks that are not part of any file; they are available to be assigned to any new or current file. FREE designates volume blocks as free by updating the volume free space map.

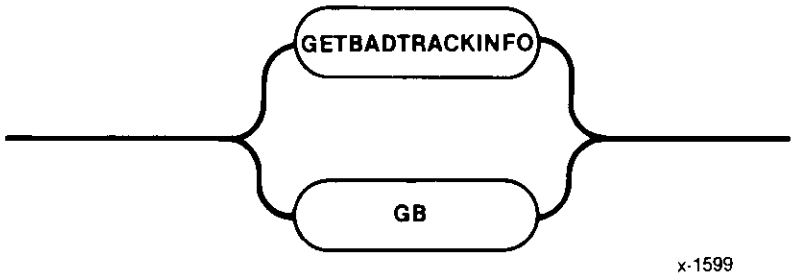
When you use the FREE command to designate one or more bad blocks as "good," it removes the block number from the bad blocks file. However, FREE BADBLOCK does not designate the blocks as free. To update the volume free space map and designate these blocks as free, use the FREE BLOCK command.

ERROR MESSAGES

argument error	A syntax error was made in the command or a nonnumeric character was specified in the blocknum or fnodenum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.
<fnodenum>, fnode out of range	The fnode number specified was larger than the largest fnode number in the volume.

no badblocks file	The volume does not have a bad blocks file. This message could appear because an earlier version of the Human Interface FORMAT command was used when the disk was formatted.
not a named disk	FREE was performed on a physical volume.

This command displays the volume's bad track information. It can be aborted by entering CONTROL-C. The format of GETBADTRACKINFO is



INPUT PARAMETERS

None.

OUTPUT

The GETBADTRACKINFO command displays the volume's bad track information as written by the manufacturer or the Human Interface FORMAT command. The output displayed by the GETBADTRACKINFO command is compatible with the format required by the Human Interface FORMAT command when writing bad track information on the disk. To use the output as input to FORMAT, remove the first two lines, leaving only the actual bad track information data. The display is as follows:

```
Bad track information:
cyl    head   sector
cccc   hh     ss
cccc   hh     ss
.      .      .
.      .      .
```

where cccc is cylinder number, hh is the head number and ss is the sector number (always zero for all devices supported in this release of the Operating System).

GETBADTRACKINFO

As mentioned above, the output of the GETBADTRACKINFO command can be used as input to the FORMAT command when creating the bad track information file. The example below shows how to use GETBADTRACKINFO this way.

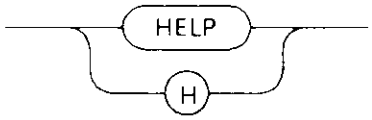
```
-attachdevice wmfdx0 as :w: <CR>
-diskverify :sd: to :w:bad.list   <CR>
*getbadtrackinfo  <CR>
*exit  <CR>
```

After exiting DISKVERIFY and rebooting the system, edit :w:bad.lst and remove the header lines. The file can then be used as input to the bad track information file created by the FORMAT command.

ERROR MESSAGES

I/O error while trying to read bad track information	An I/O error occurred while reading the bad track information.
No valid bad track info found	Bad track information is not valid and cannot be displayed.
No bad track info found	The area designated for bad track information is empty.

This command lists all available Disk Verification Utility commands and provides a short description of each command. The format of the HELP command is



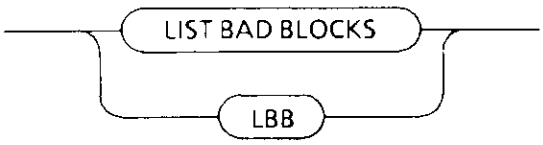
F 0214

OUTPUT

In response to this command, HELP displays the following information:

```
*help
                allocate/free : allocate/free fnodes, space blocks, bad blocks
 backup/restore fnodes (bf/rf) : backup/restore fnode file to/from save area
                   Control C : abort the command in progress
                        disk : display disk attributes
   display byte/word (d,db/dw) : display the buffer in (byte/word format)
       display directory (dd) : display the directory contents
           display fnode (df) : display fnode information
   display next block (>,dnb) : read and display 'next' volume block
display previous block (<,dpb) : read and display 'previous' volume block
     display save fnode (dsf) : display saved fnode information
                  exit,quit : quit disk verify
        list bad blocks (lbb) : list bad blocks on the volume
                   read (r) : read a disk block into the buffer
   restore volume label (rvl) : copy volume label from save area
                        save : save free fnodes, free space & bad block maps
substitute byte/word (s,sb/sw) : modify the buffer (byte/word format)
                     verify : verify the disk
                  write (w) : write to the disk block from the buffer
             edit fnode (ef) : edit an fnode
        edit save fnode (esf) : edit a saved fnode
                        fix : perform various fixes on the volume
      get bad track info (gb) : get the bad track info on the volume
misc commands-
                    address : convert block number to absolute address
                      block : convert absolute address to block number
                    hex/dec : display number as hexadecimal/decimal number
   add,+,sub,-,mul,*,div,/,mod : arithmetic operations on unsigned numbers
```

This command displays all the bad blocks on a named volume. You can abort this command by typing a CONTROL-C. The format of the LISTBADBLOCKS command is as follows:



F-0208

OUTPUT

In response to this command, LISTBADBLOCKS displays up to eight columns of block numbers that you specified as "bad." Figure 2-2 illustrates the format of the display.

```
Badblocks on Volume:  volumenum

<blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>

<blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>
    .           .           .           .           .           .
    .           .           .           .           .           .
    .           .           .           .           .           .
<blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>
```

Figure 2-2. LISTBADBLOCKS Format

If none of the blocks have been marked as "bad", LISTBADBLOCKS displays the following message:

```
no badblocks
```

NOTE

Bad tracks and bad blocks are different. Bad tracks are handled by the device drivers in conjunction with the hardware, whereas, bad blocks are handled by the iRMX II Basic I/O System.

ERROR MESSAGES

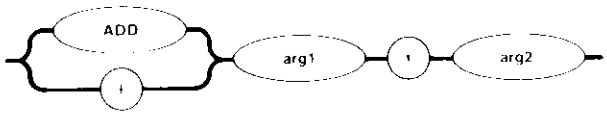
no badblocks file	The volume does not have a bad blocks file. This message could appear because an earlier version of the Human Interface FORMAT command was used when the disk was formatted or because the disk is a physical volume.

The following commands provide you with the ability to perform arithmetic and conversion operations within the Disk Verification Utility. The commands perform the operations on unsigned numbers only and do not report any overflow conditions. When the number is displayed in both hexadecimal and decimal format, it appears in hexadecimal format first, followed by the decimal number in parentheses. For example:

```
13 (   19T)
```

ADD

This command adds two numbers together. Its format is



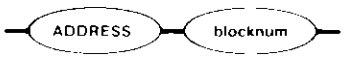
where:

arg1 and arg2 — Numbers the command adds together. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the unsigned sum of the two numbers in both hexadecimal and decimal format.

ADDRESS

All memory in a volume is divided into volume blocks, which are areas of memory the same size as the volume granularity. Volume blocks are numbered sequentially in the volume, starting with the block containing the smallest addresses (block 0). The ADDRESS command converts a block number into an absolute address (in hexadecimal) on the volume, so that you don't have to perform this conversion by hand. The format of this command is



where:

blocknum — Volume block number that ADDRESS converts into an absolute address in hexadecimal. This parameter can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume.

In response, ADDRESS displays the following information:

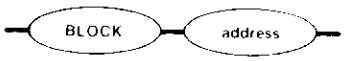
```
absolute address = <addr>
```

where:

<addr> — Absolute address in hexadecimal that corresponds to the specified block number. This address represents the number of the byte that begins the block and can range from 0 through (volume size - 1), where volume size is the size, in bytes, of the volume.

BLOCK

The BLOCK command is the inverse of the address command. It converts a 32-bit absolute address (in hexadecimal) into a volume block number, so that you don't have to perform this conversion by hand. The format of this command is



where:

address — Absolute address in hexadecimal that BLOCK converts into a block number. This parameter can range from 0 through (volume size - 1), where volume size is the size, in bytes, of the volume.

In response, BLOCK displays the following information:

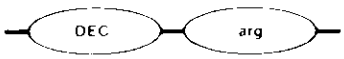
```
block number = <blocknum>
```

where:

<blocknum> — Number of the volume block that contains the specified absolute address in hexadecimal. The BLOCK command determines this value by dividing the absolute address by the volume block size and truncating the result.

DEC

This command finds the decimal equivalent of a number. Its format is



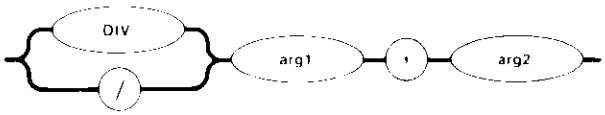
where:

arg	Number for which the command finds the decimal equivalent. The value of the argument cannot be greater than 2^{32}-1. The default base is in hexadecimal.

In response, the command displays the decimal equivalent of the specified number.

DIV

This command divides one number by another. Its format is



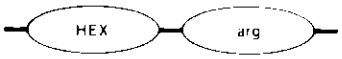
where:

arg1 and arg2	Numbers on which the command operates. It divides arg1 by arg2. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the unsigned integer quotient in both hexadecimal and decimal format.

HEX

This command finds the hexadecimal equivalent of a number. Its format is



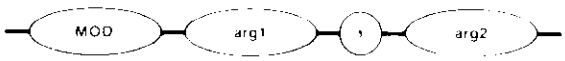
where:

arg	Number for which the command finds the hexadecimal equivalent. If you are specifying a decimal number, you must specify a "T". The value of the argument cannot be greater than 2^{32}-1.

In response, the command displays the hexadecimal equivalent of the specified number.

MOD

This command finds the remainder of one number divided by another. Its format is



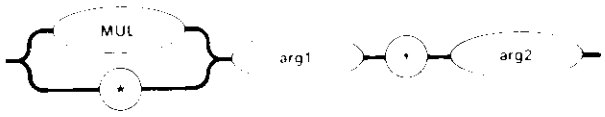
where:

arg1 and arg2	Numbers on which the command operates. It performs the operation arg1 modulo arg2. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the value arg1 modulo arg2 in both hexadecimal and decimal format.

MUL

This command multiplies two numbers together. Its format is



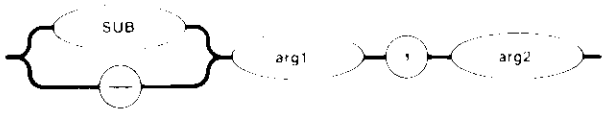
where:

arg1 and arg2 — Numbers the command multiplies together. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the unsigned product of the two numbers in both hexadecimal and decimal format.

SUB

This command subtracts one number from another. Its format is



where:

arg1 and arg2 — Numbers on which the command operates. The command subtracts arg2 from arg1. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the unsigned difference in both hexadecimal and decimal format.

ERROR MESSAGES

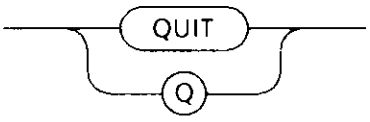
The following error messages may be returned by any of the Miscellaneous Commands:

argument error	A syntax error was made in the command, a nonnumeric value for one of the arguments was specified, or a value for a block number parameter that was not a valid block number was specified.
<blocknum>, block out of range	If the command was an ADDRESS command, the block number entered was greater than the number of blocks in the volume.
<address>, address not on the disk	If the command was a BLOCK command, BLOCK converted the address to a volume block number, but the block number was greater than the number of blocks in the volume.

EXAMPLES

```
*MUL 134T, 13T  <CR>
 6CE ( 1742T)
*+ 8, 4  <CR>
 0C  (   12T)
*SUB 8884, 256  <CR>
 862E (34350T)
*MOD 1225, 256T  <CR>
 25 (   37T)
*HEX 155T  <CR>
 9B
*ADDRESS 15  <CR>
absolute address = 0A80
*BLOCK 2236  <CR>
block number = 44
```

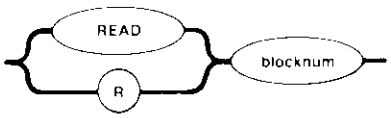
This command exits the Disk Verification Utility and returns control to the Human Interface command level. The format of the QUIT command is as follows:



F-0209

This command is identical to the EXIT command.

This command reads a volume block from the disk into the working buffer. The format of the READ command is



INPUT PARAMETER

blocknum	Number of the volume block to read. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume.

OUTPUT

In response to the command, READ reads the block into the working buffer and displays the following message:

```
read block number:  <blocknum>
```

where <blocknum> is the number of the block.

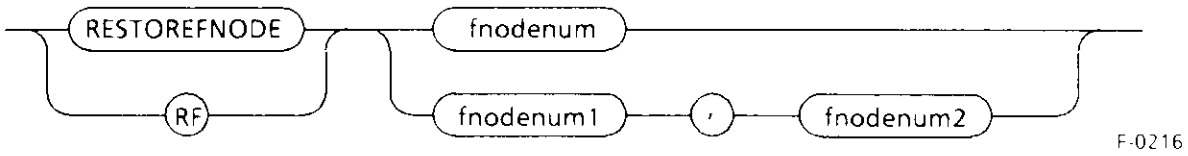
DESCRIPTION

The READ command copies a specified volume block from the volume to the working buffer. It destroys any data currently in the working buffer. Once the block is in the working buffer, you can use DISPLAYBYTE and DISPLAYWORD to display the block, and you can use SUBSTITUTEBYTE and SUBSTITUTEWORD to change the data in the block. Finally, you can use the WRITE command to write the modified block back to the volume and repair damaged volume data.

ERROR MESSAGES

argument error	A nonnumeric character was specified in the blocknum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.
FFFFFFFF, block out of range	No block number was specified and no previous read request was executed on this volume.

This command copies an fnode or a range of fnodes from the R?SAVE file to the fnode file. Before changing the fnode file, RESTOREFNODE displays the fnode number to be changed and prompts you to confirm (by entering a "y") that the fnode is to be restored. R?SAVE must have been reserved (the RESERVE option of the FORMAT command must have been specified) when the volume was formatted. The format of the RESTOREFNODE command is as follows:



INPUT PARAMETER

fnodenum	The hexadecimal number of the fnode to be restored. This number must be greater than or equal to zero and less than the maximum number of fnodes defined when the volume was formatted.
fnodenum1	The initial hexadecimal fnode number in a range of fnodes to be restored. This number must be greater than or equal to zero and less than or equal to the final fnode number in the range (fnodenum2).
fnodenum2	The final hexadecimal fnode number in a range of fnodes to be restored. This number must be greater than or equal to the initial fnode number in the range (fnodenum1) and less than the maximum number of fnodes defined when the volume was formatted.

OUTPUT

When the fnode is restored (the response to the confirmation query is "Y" or "y"):

```
restore fnode    (fnodenum)?  Y <CR>
restored fnode number:    (fnodenum)
*
```

When the fnode is not restored (the response to the confirmation query is not "Y"):

```
restore fnode     (fnodenum)?  <CR>
*
```

DESCRIPTION

The RESTOREFNODE command enables you to rebuild a damaged fnode file, thereby re-establishing links to data that would otherwise be lost. RESTOREFNODE copies an fnode or a range of fnodes from the R?SAVE file (the fnode backup file) to the fnode file. Before each of the specified fnodes is copied, RESTOREFNODE displays a query prompting you to confirm that the indicated fnode is to be restored. You must reply to this query with the letter "Y" (either "Y" or "y") to restore the fnode. If you enter any other response, RESTOREFNODE will not restore the fnode and will pass on to the next fnode in the range.

Since RESTOREFNODE operates on the R?SAVE file, you must have reserved this file when the volume was formatted. (You reserve R?SAVE by specifying the RESERVE parameter when you invoke the FORMAT command to format the volume.) If the R?SAVE file was not reserved when the volume was formatted, RESTOREFNODE will return an error message.

CAUTION

When using this command, be sure that any fnode you restore represents a file that has not been modified since the last fnode backup. RESTOREFNODE overwrites the specified fnode in the fnode file with the corresponding fnode in the R?SAVE file. If that fnode has not been backed up since the last file modification, a valid fnode may be overwritten with invalid data. Thus, all links to the associated file will be destroyed, and YOU WILL LOSE ALL OF THE DATA IN THE FILE.

ERROR MESSAGES

argument error	When the command was entered, no argument was supplied. This command requires an argument.
no save area was reserved when volume was formatted	The volume was not formatted to support backup fnodes. This means the RESERVE option was not specified when the volume was formatted.

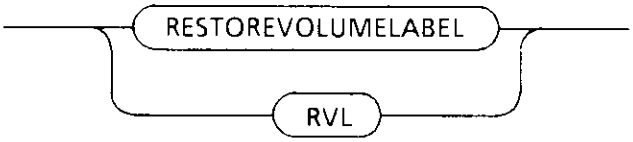
RESTOREFNODE

not a named disk	The volume specified when the Disk Verification Utility was invoked is a physical volume, not a named volume.
<fnode num>, fnode out of range	The fnode number specified is not in the range of 0 to (maximum fnodes - 1).
allocation bit not set for saved fnode restore fnode <fnode num>?	The fnode you specified has not been backed up in the R?SAVE file. If you respond to the query with a "Y", THE DATA IN THE FILE ASSOCIATED WITH THE ORIGINAL FNODE WILL BE LOST.

EXAMPLE

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
*restorefnode 9,0B <CR>   or   rf 9,0B <CR>
restore fnode     9?  Y <CR>
restored fnode number:    9
restore fnode    0A?  Y <CR>
restored fnode number:   0A
restore fnode    0B?  Y <CR>
restored fnode number:   0B
*
```

This command copies the duplicate volume label to the volume label on track 0. The duplicate volume label must have been constructed when the volume was formatted. (That is, the RESERVE option of the FORMAT command must have been specified when the volume was formatted.) The format of the RESTOREVOLUMELABEL command is as follows:



F-0211

INPUT PARAMETERS

None.

OUTPUT

volume label restored

DESCRIPTION

The RESTOREVOLUMELABEL command enables you to rebuild a damaged volume label, thereby re-establishing links to data that would otherwise be lost. RESTOREVOLUMELABEL copies the duplicate volume label to the volume label offset on track 0. When you use the Human Interface FORMAT command to create the duplicate volume label (by specifying the RESERVE parameter), the volume label is automatically copied to the end of the R?SAVE file. Because the contents of the volume label do not change, no other volume label backup is required.

If a duplicate volume label has been reserved on a volume, the Disk Verification Utility can access that volume as a Named volume even if the volume label is damaged. When the original volume label is corrupted, the Disk Verification Utility attempts to use the duplicate volume label. If the backup label is used, a "DUPLICATE VOLUME LABEL USED" message appears when the utility is invoked.

If the duplicate volume label was not reserved when the volume was formatted, RESTOREVOLUMELABEL will return an error message.

RESTOREVOLUMELABEL

ERROR MESSAGES

argument error	When the command was entered, an argument was supplied. This command does not accept an argument.
no save area was reserved when volume was formatted	The volume has not been formatted to support volume label backup.
not a named disk	The volume specified when the Disk Verification Utility was invoked is a physical volume, not a named volume.

EXAMPLE

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
DUPLICATE VOLUME LABEL USED
*restorevolumelabel <CR>  or  rvl <CR>
volume label restored
*
```

This command writes the reconstructed free fnodes bit map, volume free space bit map, and the bad blocks bit map to the volume being verified. (The NAMED2 and PHYSICAL options of the VERIFY command originally created the maps.) The format of the SAVE command is

x 24

OUTPUT

In response to this command, SAVE displays the following message:

```
save fnode map?
```

If you want to write the reconstructed free fnodes map to the volume, enter Y, y, or YES. Otherwise, enter any other character or a carriage return. If you enter YES, SAVE writes the free fnodes map to the volume and displays the following message:

```
free fnode map saved
```

In any case, SAVE next displays the following message:

```
save space map?
```

If you want to write the reconstructed free space map to the volume, enter Y or YES. Otherwise, enter any other character or a carriage return. If you enter YES, SAVE writes the volume free space map to the volume and displays the following message:

```
free space map saved
```

SAVE displays the following message if the bad blocks map is reconstructed:

```
save bad block map?
```

If you want to write the reconstructed bad blocks map to the volume, enter Y, y, or YES. Otherwise, enter any other character or a carriage return. If you enter YES, SAVE writes the volume bad blocks map to the volume and displays the following message:

```
bad block map saved
```

SAVE

DESCRIPTION

Whenever you perform a VERIFY function with the NAMED2 option (refer to the description of the VERIFY command for more information), VERIFY creates its own free fnodes map and volume free space map. It does this by examining all directories and fnodes on the volume, not by copying the maps that exist on the volume. To create the free fnodes map, it examines every directory on the volume to determine which fnodes represent actual files. To create the volume free space map, it examines the POINTER(n) fields of the fnodes to determine which volume blocks the files use.

If the volume has a bad blocks file and you perform a VERIFY function with the PHYSICAL option (refer to the description of the VERIFY command for more information), VERIFY creates its own bad blocks map. It does this by examining every block on the volume, not by copying the maps that exist on the volume.

VERIFY then compares the newly created maps with the maps that exist on the volume. If a discrepancy exists, VERIFY displays a message indicating this.

The SAVE command takes the free fnodes map, the volume free space map, and the bad block map created during the VERIFY operation and writes them to the volume, replacing the maps that currently exist.

ERROR MESSAGE

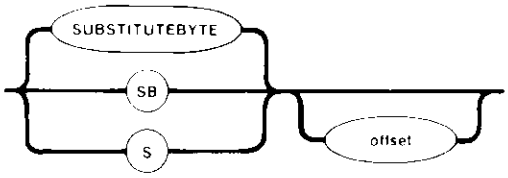
nothing to save	No bit map was constructed prior to invoking SAVE. (Bit maps are constructed by NAMED2 or PHYSICAL verifications.)

EXAMPLE

The following example illustrates the format of the SAVE command after you use VERIFY and the NAMED or NAMED2 option.

```
*VERIFY NAMED2 <CR>
'NAMED2' VERIFICATION
      .
      .
      .
   BIT MAPS O.K.
*SAVE  <CR>
save fnode map? y   <CR>
   free fnode map saved
save space map? y  <CR>
   free space map saved
*
```

This command enables you to interactively change the contents of the working buffer (in byte format). You can abort this command by typing a CONTROL-C. The format of the SUBSTITUTEBYTE command is



x742

INPUT PARAMETER

offset	Number of the first byte, relative to the start of the working buffer, that you want to change. This number can range from 0 to (block size - 1), where block size is the size of a volume block (and thus the size of the working buffer). If you omit this parameter, the command assumes a value of 0.

OUTPUT

In response to the command, SUBSTITUTEBYTE displays the specified byte and waits for you to enter a new value. This display appears as

```
<offset>: val -
```

where <offset> is the number of the byte, relative to the start of the buffer, and val is the current value of the byte. At this point, you can enter one of the following:

- A value followed by a carriage return. This causes SUBSTITUTEBYTE to substitute the new value for the current byte. If the value you enter requires more than one byte of storage, SUBSTITUTEBYTE uses only the low-order byte of the value. It then displays the next byte in the buffer and waits for further input.
- A carriage return alone. This causes SUBSTITUTEBYTE to leave the current value as is and display the next byte in the buffer. It then waits for further input.

- A value followed by a period (.) and a carriage return. This causes SUBSTITUTEBYTE to substitute the new value for the current byte. It then exits from the SUBSTITUTEBYTE command and gives the asterisk (*) prompt, enabling you to enter any DISKVERIFY command.
- A period (.) followed by a carriage return. This exits the SUBSTITUTEBYTE command and gives the asterisk (*) prompt, enabling you to enter any DISKVERIFY command.

DESCRIPTION

With the SUBSTITUTEBYTE command you can interactively change bytes in the working buffer. Once you enter the command, SUBSTITUTEBYTE displays the offset and the value of the first byte. You can change the byte by entering a new byte value, or you can leave the byte as is by entering a carriage return. The command then displays the next byte in the buffer. In this manner, you can consecutively step through the buffer, changing whatever bytes are appropriate. When you finish changing the buffer, you can enter a period followed by a carriage return to exit the command.

The SUBSTITUTEBYTE command considers the working buffer to be a circular buffer. That is, entering a carriage return when you are positioned at the last byte of the buffer causes SUBSTITUTEBYTE to display the first byte of the buffer.

The SUBSTITUTEBYTE command changes only the values in the working buffer. To make the changes in the volume, you must enter the WRITE command to write the working buffer back to the volume.

ERROR MESSAGES

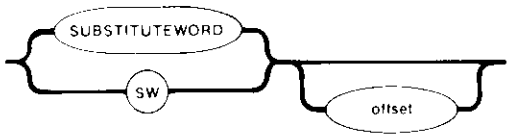
argument error	A nonnumeric character was specified in the offset parameter.
<offsetnum>, invalid offset	An offset value larger than the number of bytes in the block was specified.

EXAMPLE

This example changes several bytes in two portions of the working buffer. Two SUBSTITUTEBYTE commands are used.

```
*SUBSTITUTEBYTE<CR>

0000: A0 - 00<CR>
0001: 80 - <CR>
0002: E5 - <CR>
0003: FF - 31<CR>
0004: FF - .<CR>

*SUBSTITUTEBYTE 40<CR>

0040: 00 - E6<CR>
0041: 00 - E6<CR>
0042: 00 - .<CR>

*
```

This command is identical to SUBSTITUTEBYTE, except that it displays the buffer in WORD format, and substitutes word values in the buffer. The format of the SUBSTITUTEWORD command is



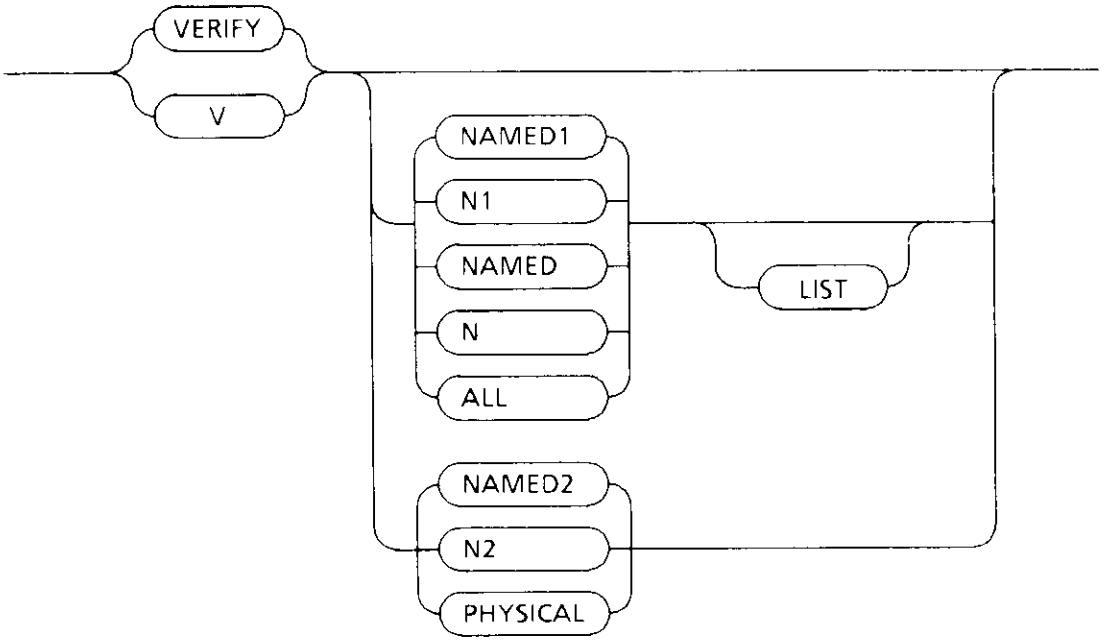
x 243

EXAMPLE

This example changes several bytes in two areas of the working buffer. Two SUBSTITUTEWORD commands are used. In the first command the words begin on even addresses, and in the second command, they begin on odd addresses.

```
*SUBSTITUTEWORD<CR>

0000: A0B0 - 0000<CR>
0002: 8070 - <CR>
0004: E511 - <CR>
0006: FFFF - 3111<CR>
0008: FFFF - .<CR>

*SUBSTITUTEWORD 35<CR>

0035: 0000 - E6FF<CR>
0037: 0000 - E6AB<CR>
0039: 0000 - .<CR>

*
```

This command checks the structures on the volume to determine whether the volume is properly formatted. You can abort this command by typing a CONTROL-C. The format of the VERIFY command is



F-0217

INPUT PARAMETERS

NAMED1 or N1 Checks named volumes to ensure that the information recorded in the fnodes is consistent and matches the information obtained from the directories themselves. VERIFY performs the following operations during a NAMED1 verification:

- Checks fnode numbers in the directories to see if they correspond to allocated fnodes.
- Checks the parent fnode numbers recorded in the fnodes to see if they match the information recorded in the directories.
- Checks the fnodes against the files to determine if the fnodes specify the proper file type.
- Checks the POINTER(n) structures of long files to see if the indirect blocks accurately reflect the number of blocks used by the file.
- Checks each fnode to see if the TOTAL SIZE, TOTAL BLKS, and THIS SIZE fields are consistent.
- Checks the bad blocks file to see if the blocks in the file correspond to the blocks marked as "bad" on the volume.
- Checks the checksum of each fnode.

NAMED2 or N2	Checks named volumes to ensure that the information recorded in the free fnodes map and the volume free space map matches the actual files and fnodes. VERIFY performs the following operations during a NAMED2 verification: • Creates a free fnodes map by examining every directory in the volume. It then compares that free fnodes map with the one already on the volume. • Creates a free space map by examining the information in the fnodes. It then compares that free space map with the one already on the volume. • Checks to see if the block numbers recorded in the fnodes and the indirect blocks actually exist. • Checks to see if two or more files use the same volume block. If so, it lists the files referring to each block. • Checks the volume free space map for any bad blocks that are marked as "free." • Checks to see if two or more directories reference the same fnode. If so, it lists the directories referring to each fnode.
NAMED or N	Performs both the NAMED1 and NAMED2 operations on a named volume. If you specify the VERIFY command with no option, NAMED is the default.
ALL	Performs all operations appropriate to the volume. For named volumes, this option performs both the NAMED and PHYSICAL operations. For physical volumes, this option performs only the PHYSICAL operations.
PHYSICAL	Reads all blocks on the volume and checks for I/O errors. This parameter applies to both named and physical volumes. VERIFY also creates a bad blocks map by examining every block on the volume.
LIST	When you specify this option, the file information in Figure 2-3 is displayed for every file on the volume, even if the file contains no errors. You can use this option with all parameters that, either explicitly or implicitly, specify the NAMED1 parameter.

OUTPUT

VERIFY produces a different kind of output for each of the NAMED1, NAMED2, and PHYSICAL options. The NAMED and ALL options produce combinations of these three kinds of output.

Figure 2-3 illustrates the format of the NAMED1 output (without the LIST option).

```
DEVICE NAME = <devname>  : DEVICE SIZE = <devsize>  : BLOCK SIZE = <blksize>

'NAMED1' VERIFICATION

FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
    <error messages>

FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
    <error messages>
   .          .          .          .          .          .
   .          .          .          .          .          .
   .          .          .          .          .          .
FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
    <error messages>
```

Figure 2-3. NAMED1 Verification Output

The following paragraphs identify the fields listed in Figure 2-3.

<devname>	Physical name of the device, as specified in the ATTACHDEVICE Human Interface command.
<devsize>	Hexadecimal size of the volume, in bytes.
<blksize>	Hexadecimal volume granularity. This number is the size of a volume block.
<filename>	Name of the file (1 to 14 characters).
<fnodenum>	Hexadecimal number of the file's fnode.
<lev>	Hexadecimal level of the file in the file hierarchy. The root directory of the volume is the only level 0 file. Files contained in the root directory are level 1 files. Files contained in level 1 directories are level 2 files. This numbering continues for all levels of files in the volume.
<parnt>	Fnode number of the directory that contains this file, in hexadecimal.

<typ>	File type, either DATA (data files), DIR (directory files), SMAP (volume free space map), FMAP (free fnodes map), BMAP (bad blocks map), or VLAB (volume label file). If VERIFY cannot ascertain that the file is a directory or data file, it displays the characters "*****" in this field.
<error messages>	Messages that indicate the errors associated with the previously-listed file. The possible error messages are listed later in this section.

As Figure 2-3 shows, the NAMED1 option (without the LIST option) displays information about each file that is in error. If you used the LIST option with the NAMED1 option, the file information in Figure 2-3 is displayed for every file, even if the file contains no errors. The NAMED1 display also contains error messages that immediately follow the list of the affected files.

Figure 2-4 illustrates the format of the NAMED2 output. If VERIFY detects an error during NAMED2 verification, it displays one or more error messages in place of the "BIT MAPS O.K." message.

```
DEVICE NAME = <devname> : DEVICE SIZE = <devsize> : BLK SIZE = <blksze>

'NAMED2' VERIFICATION

   BIT MAPS O.K.
```

Figure 2-4. NAMED2 Verification Output

The fields in Figure 2-4 are exactly the same as the corresponding fields in Figure 2-3.

Figure 2-5 illustrates the format of the PHYSICAL output.

```
DEVICE NAME = <devname>    : DEVICE SIZE = <devsize> : BLOCK SIZE = <blksize>

'PHYSICAL' VERIFICATION

        NO ERRORS
```

Figure 2-5. PHYSICAL Verification Output

The fields in Figure 2-5 are exactly the same as the corresponding fields in Figure 2-3.

If VERIFY detects an error during PHYSICAL verification, it displays the message:

```
<blocknum>, error
```

in place of the "NO ERRORS" message.

If you specify NAMED verification, VERIFY displays both the NAMED1 and NAMED2 output. If you specify the ALL verification for a named volume, VERIFY displays the NAMED1, NAMED2, and PHYSICAL output. If you specify the ALL verification for a physical volume, VERIFY displays the PHYSICAL output.

DESCRIPTION

The VERIFY command checks physical and named volumes to ensure that the volumes contain valid file structures and data areas. VERIFY can perform three kinds of verification: NAMED1, NAMED2, and PHYSICAL. NAMED1 and NAMED2 verifications check the file structures of named volumes. They do not apply to physical volumes. A PHYSICAL verification checks each data block of the volume for I/O errors. PHYSICAL verification applies to both named and physical volumes.

As part of the NAMED2 verification, VERIFY creates a free fnodes map and a volume free space map, which it compares with the corresponding maps on the volume. You can use the SAVE command to write the maps produced during NAMED2 verification to the volume, overwriting the maps on the volume.

When you perform a PHYSICAL verification on a named volume, VERIFY also creates a bad blocks map. You can use the SAVE command to write the bad blocks map produced during PHYSICAL verification to the volume; this destroys the bad blocks map already on the volume.

ERROR MESSAGES

Four kinds of error messages can occur as a result of entering the VERIFY command: VERIFY command errors, NAMED1 errors, NAMED2 errors, and PHYSICAL errors.

VERIFY Command Error

argument error — The parameter specified is not a valid VERIFY parameter.

NAMED1 Messages

The following messages can appear in a NAMED1 display, immediately after the file to which they refer.

- <blocknum 1 - blocknum n>, block bad

  The block numbers displayed in this message are marked as "bad."

- <blocknum 1 - blocknum n >, invalid block number recorded in the

  fnode/indirect block

  One of the POINTER(n) fields in the fnode specifies block numbers larger than the largest block number in the volume.

- directory stack overflow

  This message indicates that a directory on the volume lists, as one of its entries, itself or one of the parent directories in its pathname. If this happens, the utility, when it searches through the directory tree, continually loops through a portion of the tree, overflowing an internal buffer area. In this case, performing NAMED2 verification may indicate the cause of this problem.

- file size inconsistent

  total$size = <totsize> :this$size = <thsize> :data blocks = <blks>

  The TOTAL SIZE, THIS SIZE, and TOTAL BLKS fields of the fnode are inconsistent.

- <filetype>, illegal file type

  The file type of a user file, as recorded in the TYPE field of the fnode, is not valid. The valid file types and their descriptions are as follows:

File type	Number	Description
SMAP	1	volume free space map
FMAP	2	free fnodes map
BMAP	4	bad blocks map
DIR	6	directory
DATA	8	data
VLAB	9	volume label file

- <fnodenum>, allocation status bit in this fnode not set

  The file is listed in a directory but the flags field of its fnode indicates that fnode is free. The free fnodes map may or may not list the fnode as allocated.

- <fnodenum>, fnode out of range

  The fnode number is larger than the largest fnode number in the fnode file.

- <fnodenum>, parent fnode number does not match

  The file represented by fnodenum is contained within a directory whose fnode number does not match the parent field of the file.

- invalid blocknum recorded in the fnode/indirect block

  One of the pointers within the fnode or within the indirect block specifies a block number that is larger than the largest block number in the volume.

- insufficient memory to create directory stack

  There is not enough dynamic memory available in the system for the utility to perform the verification.

- sum of the blks in the indirect block does not match block in the fnode

  The file is a long file, and the number of blocks listed in a POINTER(n) field of the fnode does not agree with the number of blocks listed in the indirect block.

- total-blocks does not reflect the data-blocks correctly

  The TOTAL BLKS field of the fnode and the number of blocks recorded in the POINTER(n) fields are inconsistent.

- Bad Checksum, checksum is : <number>

  Checksum should be : <number>

  An invalid checksum has been calculated.

NAMED2 Messages

The following messages can appear in a NAMED2 display.

- <blocknum1 - blocknum2>, bad block not allocated

  The volume free space map indicates that the blocks are free, but they are marked as "bad" in the bad blocks file.

- <blocknum>, block allocated but not referenced

  The volume free space map lists the specified volume block as allocated, but no fnode specifies the block as part of a file.

- <blocknum>, block referenced but not allocated

  An fnode indicates that the specified volume block is part of a file, but the volume free space map lists the block as free.

- directory stack overflow

  This message can indicate that a directory on the volume lists, as one of its entries, itself or one of the parent directories in its pathname. If this happens, the utility, when it searches through the directory tree, continually loops through a portion of the tree, overflowing an internal buffer area. The "Multiple Reference" message (explained below) may help you find the cause of this problem.

- Fnodes map indicates fnodes > max$fnode

  The free fnodes map indicates that there are a greater number of unallocated fnodes than the maximum number of fnodes in the volume.

- <fnodenum>, fnode-map bit marked allocated but not referenced

  The free fnodes map lists the specified fnode as allocated, but no directory contains a file with the fnode number.

- <fnodenum>, fnode referenced but fnode-map bit marked free

  The specified fnode number is listed in a directory, but the free fnodes map lists the fnode as free.

- Free space map indicates Volume block > max$Volume$block

  The free space map indicates that there are a greater number of unallocated blocks than the maximum number of blocks in the volume.

- insufficient memory to create directory stack

  Not enough dynamic memory is available in the system for the utility to perform the verification.

- insufficient memory to create fnode and space maps

  During a NAMED2 verification, the utility tried to create a free fnodes map and a volume free space map. However, not enough dynamic memory is available in the system to create these maps.

- insufficient memory to create bad blocks map

  During a PHYSICAL verification, the utility tried to create a bad blocks map. However, not enough dynamic memory is available in the system to create the map.

- Multiple reference to fnode <fnodenum>
  Path name : <full path name>
  referring fnodes:
  <fnodenum> Path name: <full path name>
  <fnodenum> Path name: <full path name>

  The directories on the volume list more than one file associated with this fnode number.

- Multiple reference to block <blocknum>
  referring fnodes:
  <fnodenum> Path name: <full path name>
  <fnodenum> Path name: <full path name>

  More than one fnode specifies this block as part of a file.

PHYSICAL Messages

- <blocknum>, error

  An I/O error occurred when VERIFY tried to access the specified volume block. The volume is probably flawed.

Miscellaneous Messages

The following messages indicate internal errors in the Disk Verification Utility. Under normal conditions these messages should never appear. However, if these messages (or other undocumented messages) do appear during a NAMED1 or NAMED2 verification, you should exit the Disk Verification Utility and re-enter the DISKVERIFY command.

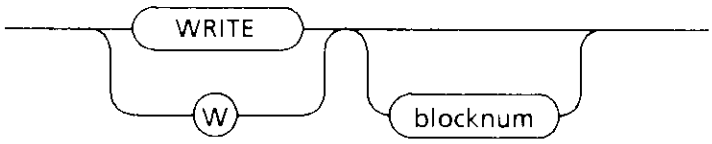
directory stack empty
directory stack error
directory stack underflow

EXAMPLE

The following command performs both named and physical verification on a named volume.

```
*VERIFY ALL <CR>

DEVICE NAME = F1            : DEVICE SIZE = 0003E900 : BLOCK SIZE = 0080

'NAMED1' VERIFICATION

'NAMED2' VERIFICATION
   BIT MAPS O.K.
'PHYSICAL' VERIFICATION
   NO ERRORS
*
```

This command writes the contents of the working buffer to the volume. The format of this command is



F-0210

INPUT PARAMETER

blocknum	Number of the volume block to which the command writes the working buffer. If you omit this parameter, WRITE writes the buffer back to the block most recently accessed.

OUTPUT

In response to the command, WRITE displays the following message:

```
write to block <blocknum>?
```

where <blocknum> is the number of the volume block to which WRITE intends to write the working buffer. If you respond by entering Y or any character string beginning with Y or y, WRITE copies the working buffer to the specified block on the volume and displays the following message:

```
written to block number:<blocknum>
```

Any other response aborts the write process.

DESCRIPTION

The WRITE command is used in conjunction with the READ, DISPLAYBYTE, DISPLAYWORD, SUBSTITUTEBYTE, and SUBSTITUTEWORD commands to modify information on the volume. Initially you use READ to copy a volume block from the volume to a working buffer. Then you can use DISPLAYBYTE and DISPLAYWORD to view the buffer and SUBSTITUTEBYTE and SUBSTITUTEWORD to change the buffer. Finally, you can use WRITE to write the modified buffer back to the volume. By default, WRITE copies the buffer to the block most recently accessed by a READ or WRITE command.

A WRITE command does not destroy the data in the working buffer. The data remains the same until the next SUBSTITUTEBYTE, SUBSTITUTEWORD, or READ command modifies the buffer.

ERROR MESSAGES

argument error	A syntax error was made or nonnumeric characters were specified in the blocknum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.
FFFFFFFF, block out of range	No blocknum was specified and no previous read request was executed on this volume.

EXAMPLE

The following command copies the working buffer to the block from which it was read.

```
*WRITE  <CR>
write 4B?  y <CR>
written to block number: 4B
*
```

CHAPTER 3
BACKING UP AND RESTORING FNODES

3.1 INTRODUCTION

To access data on a named volume (such as a disk), the iRMX II Operating System uses a mechanism common to virtually all operating systems: it maintains an index to every file on the disk. This index is created when the disk is formatted and remains as a permanent structure at a dedicated location on the disk. The index consists of a system of pointers that indicate the location of the data files on the disk. Thus, when data must be stored on or retrieved from the disk, the operating system can find the exact location of the appropriate file by looking up the file name in the index.

In the operating system, the index consists of the iRMX II volume label and an fnode file. This volume label resides at the same location in all devices and serves as the initial entry point into the device. The fnode file can reside anywhere on the disk (specified when the disk is formatted) and contains a series of individual structures called file descriptor nodes or "fnodes." There is one fnode for each file on the disk. The fnode contains information essential to accessing and maintaining the respective file.

The iRMX II file structure for a named volume is organized as a hierarchical tree. That is, there is a root directory with branches to other directories and ultimately, to files. The organization of the fnode file reflects this hierarchical structure. The iRMX II volume label contains a pointer to the fnode of the file structure's root directory. The root directory is always the starting address for any file or directory on the volume. It lists all the first level files and directories on the volume. First level directories point to second level files and directories, and so on, down the hierarchical structure.

As previously mentioned, each file or directory is represented by an fnode. The fnode, along with other data describing the file or directory, contains pointers to blocks on the volume. If the fnode describes a short file, these blocks contain the actual file data. If the fnode describes a long file, these blocks contain pointers to other blocks containing the actual data. If the fnode describes a directory, these blocks contain entries which describe the contents of the directory. Each entry lists the fnode number and name of the associated file or directory.

The operating system creates the iRMX II volume label and the fnode file when the disk is formatted.

The number of unallocated fnodes in the fnode file is controlled by the FILES parameter of the FORMAT command. In addition to the unallocated fnodes, seven (with an option of eight) allocated fnodes are established when the fnode file is created. These allocated fnodes represent

- the fnode file
- the volume label file - R?VOLUMELABEL
- the volume free space map file - R?SPACEMAP
- the free fnodes map file - R?FNODEMAP
- the bad blocks file - R?BADBLOCKMAP
- the root directory
- the space accounting file,
- Optionally, the duplicate volume label file - R?SAVE

For a full description of these files, see Appendix A "Structure of A Named Volume."

Thereafter, when files or directories are created directly subordinate to the root, the operating system must adjust a pointer in the root fnode to indicate the fnode number of the new data file or directory file. Subsequently, directories subordinate to the root must also have their pointers adjusted when they become parents to a new data file or directory.

This method of storing and retrieving data on a disk has one major drawback. All access to files on the disk is through the iRMX II volume label and the fnode file. If either the volume label file or the fnode file is damaged or destroyed, there is no practical way to recover data on the disk.

The backup and restore fnodes feature enables some recovery of data lost as a result of damage to the fnode file or the iRMX II volume label. With this feature, you create a backup version of the volume label and all the fnodes on the disk. The backup version is stored in one of the innermost tracks of the disk where the chance of accidental loss of data is minimal. (In normal use, the disk heads do not extend to the innermost tracks.)

To implement this feature, the Human Interface FORMAT command has been modified to include an optional parameter -- RESERVE. This version of the FORMAT command creates a file named R?SAVE in the innermost track of the volume. A copy of the iRMX II volume label is placed in the front (that is, the physical end) of the file and an fnode is allocated for R?SAVE in the fnode file. (The fnode for the R?SAVE file is allocated out of the fnodes reserved through the FILES parameter of the FORMAT command. Thus, if you specify "FILES = 3000" when you format, only 2999 of those fnodes will remain available after the R?SAVE fnode has been allocated.) Finally, FORMAT copies the fnode file into R?SAVE.

Notice that the FORMAT command creates a backup of the fnode file in its initialized state. R?SAVE is not subsequently updated as files are written to or deleted from the volume. Therefore, you will have to use the BACKUPFNODES Disk Verification Utility command or the BACKUP option of the Human Interface SHUTDOWN command to back up the fnode file at regular intervals. If the volume label or the fnode file become damaged, you can attempt to recover files on the volume by using the Disk Verification Utility commands (RESTOREFNODE and RESTOREVOLUMELABEL) to rebuild the index. To assist in this process, the DISPLAYSAVEFNODE Disk Verification Utility command enables you to look at individual fnodes stored in the R?SAVE file.

Since the contents of the iRMX II volume label do not change, the copy of the volume label in R?SAVE is always valid. Therefore, you can restore the volume label at any time regardless of when the R?SAVE file was last updated. (When the Disk Verification Utility encounters a damaged volume label, it automatically uses the backup volume label if the R?SAVE file is present, however, it does not restore unless explicitly instructed.)

CAUTION

One note of caution: The fnode file is changed each time a volume is modified (that is, each time a file or directory is created, written to, or deleted from the volume). Therefore, valid restoration can be assured only for fnodes whose associated files or directories have not been changed since the last backup.

If the fnodes are not backed up after each modification, the structure of the R?SAVE file will differ from that of the fnode file. Some fnodes in R?SAVE may not be associated with the same files as the corresponding fnodes in the fnode file. Attempting to recover fnodes under these conditions is dangerous because the RESTOREFNODE command will overwrite what may be a valid fnode in the fnode file.

While the backup and restore fnodes feature is a useful aid in attempting to recover data on a volume, this capability is limited in scope. If you are troubleshooting your system, you may want to back up the fnodes on the system disk before taking any action that may risk the disk's integrity. You may also decide to back up the fnodes on a routine basis (before or during each system shutdown, for instance) so that the R?SAVE file is always relatively current. However, under normal circumstances, where a volume is accessed and modified frequently, backing up the fnodes after each modification is not practical. The most practical solution is to back up the fnode file once a day using the BACKUP option of the SHUTDOWN command.

Note that this feature is not intended to provide comprehensive protection from the loss of data associated with damaged iRMX II volume labels or fnode files. Rather, it offers a tool that, when properly applied, can be useful in maintaining volume integrity in certain situations. For comprehensive protection against loss of data use the Human Interface BACKUP command.

3.2 USING FNODE BACKUP AND RESTORE

To use the fnode backup and restore feature, you must use Version 1.1 (or later) of the Human Interface FORMAT command and the Version 2.0 (or later) of the Disk Verification Utility. Used together, these versions of the FORMAT command and the Disk Verification Utility enable you to

- format a volume to create the backup file (R?SAVE)
- back up the fnodes of any files written to the volume
- examine the contents of the backup file (R?SAVE)
- restore damaged fnodes
- restore the volume label
- edit fnodes or save fnodes

This section describes how to perform each of these operations. A brief overview of the operation is followed by one or more examples of a typical implementation. In the examples, blue or bolded text indicates an entry you make from your terminal. Standard type (this is standard type) indicates system output to your terminal.

3.2.1 Creating the R?SAVE Fnode Backup File

If you intend to backup the volume label and the fnodes on a volume, you must first create the R?SAVE backup file on the innermost tracks of the volume. To do so, you must invoke Version 2.0 of the Human Interface FORMAT command, specifying the RESERVE option. NOTE THAT THE FORMAT COMMAND OVERWRITES ALL OF THE DATA CURRENTLY ON THE DISK. Therefore, make a backup copy of any files you wish to save using the Human Interface BACKUP command.

Once the volume has been formatted, the R?SAVE file will contain a copy of the fnode file including the allocated fnodes (R?SPACEMAP, R?FNODEMAP, etc.). Therefore, you need not back up the fnode file immediately after formatting the volume.

PROCEDURE

From the Human Interface, invoke the FORMAT command, specifying the RESERVE parameter.

EXAMPLE

Assume that you have booted your system from a flexible diskette to format the system disk. The command listed below formats the disk and creates the R?SAVE backup file. The initialized fnode file is copied into R?SAVE.

```
-attachdevice cmbo as :mydisk: <CR>
-format :mydisk: il = 4 files = 3000 reserve <CR>

volume ( ) will be formatted as a NAMED volume
    granularity        =  1,024 map start   =   7,859
    interleave         =      4
    files              =  3000
    extensionsize      =      3
    save area reserved =    yes
    bad track/sector information written = no
    volume size        = 15,984K

TTTTTTTTTTTTTTTT
volume formatted
```

The disk has now been formatted. A file named R?SAVE has been reserved in the innermost tracks of the disk. (If you use the Disk Verification Utility DISPLAYDIRECTORY command on the volume root fnode (fnode 6) or the Human Interface DIR command with the invisible (I) option on the volume root directory, you will find an fnode listed for R?SAVE.) R?SAVE contains a duplicate copy of the fnodes in the fnode file. That is, R?SAVE contains eight allocated fnodes (R?SAVE, R?SPACEMAP, R?FNODEMAP, etc.) and 2,999 unallocated fnodes. (Remember, the R?SAVE fnode is allocated out of the 3,000 fnodes specified through the FILES parameter.)

3.2.2 Backing up Fnodes on a Volume

DESCRIPTION

To back up the fnodes on a volume, you must have previously reserved the back up file R?SAVE when the volume was formatted. Thereafter, any modification to the volume (creating, writing to, or deleting a file) requires that the fnodes be backed up if the R?SAVE file is to contain an accurate copy of the fnode file.

You can backup the fnode on a volume either by:

- Using the Human Interface SHUTDOWN command with the BACKUP option
- Using the BACKUPFNODES option of DISKVERIFY (see Chapter 2)

EXAMPLE 1

This example shows how to backup the fnode file using SHUTDOWN with the BACKUP option. The BACKUP option allows you to copy the volume fnode file to its duplicate file, R?SAVE, on any attached volume.

```
super-SHUTDOWN  B  <CR>
***SYSTEM WILL BE SHUTDOWN IN 10 MINUTE(S)
:SD:, outstanding connections to device have been deleted
***SHUTDOWN COMPLETED  ***
```

R?SAVE now contains a duplicate copy of all fnodes in the fnode file.

EXAMPLE 2

This example shows how to use the BACKUPFNODE command of DISKVERIFY to backup the fnode file. Assume that the system disk is attached as logical device :sd:. The initial contents of the :sd: fnode file were copied to R?SAVE by the FORMAT command. A file has just been written to the volume. An fnode for this file is entered in the fnode file; however, no corresponding entry has been made in R?SAVE. The following sequence of commands will copy all fnodes in the fnode file into the R?SAVE file.

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
*backupfnodes <CR>    or    bf <CR>
fnode file backed up to save area
*
```

R?SAVE now contains a duplicate copy of all fnodes (allocated and unallocated) in the fnode file.

Note that in both cases you must reboot the system after backing up the fnodes on the volume.

3.2.3 Backing up the Volume Label

The volume label is initially copied to R?SAVE when the volume is formatted. Since the contents of the volume label do not change, no other volume label backup procedures are required.

3.2.4 Restoring Fnodes

DESCRIPTION

To restore fnodes on a volume, you must have previously reserved the backup file R?SAVE when the volume was formatted. If damage has occurred to the fnode file, you can attempt to rebuild the file (or portions of it) by using the Disk Verification Utility RESTOREFNODE command.

RESTOREFNODE enables you to restore a single fnode or a range of fnodes. You designate the fnodes to be restored by entering the fnode numbers. The specified fnodes in R?SAVE are copied into the corresponding fnodes in the fnode file.

Before restoring each fnode, RESTOREFNODE prompts you with the message "restore fnode <fnode number>?". To restore the fnode, you must enter "yes" or the letter "Y" (either Y or y). If you enter any other response, the fnode will not be restored.

When restoring fnodes, you must be very careful to ensure that you are not overwriting a valid fnode in the fnode file with an invalid fnode from R?SAVE. Be sure the volume has not been modified since the fnodes were last backed up.

PROCEDURE

1. Invoke the Disk Verification Utility, using the logical device name of the volume to be backed up.
2. When you receive the Disk Verification Utility prompt (*), enter the appropriate Disk Verification Utility commands (VERIFY, DISPLAYFNODE, etc.) to examine the fnodes file and determine which fnode must be restored.
3. Invoke the Disk Verification Utility RESTOREFNODE command to replace the damaged fnodes. The Disk Verification Utility prompts you to confirm that the proper fnode is being restored. Make sure you have specified the correct hexadecimal number for the fnode, then enter the letter "Y" in response to the prompt.
4. RESTOREFNODE returns the message "restored fnode < fnode number >" after the fnode in the R?SAVE file has been written over the corresponding fnode in the fnode file.

EXAMPLE 1

Assume that a disk drive is attached as logical device :sd:. The volume :sd: contains the R?SAVE fnode backup file. You have not modified the disk since the fnodes were last backed up. You suspect the fnode file has been damaged, so you use the Disk Verification Utility to confirm your suspicions:

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
*verify
    .
    .
    .
```

After using the Disk Verification Utility to examine the structure of the disk, you find that fnodes 9 through 0C have probably been destroyed. You then use the RESTOREFNODE command to recover these fnodes.

```
*restorefnode 9, 0C <CR>    or   rf 9, 0C <CR>
restore fnode     9?  Y <CR>
restored fnode number:    9
restore fnode    0A?  Y <CR>
restored fnode number:   0A
restore fnode    0B?  Y <CR>
restored fnode number:   0B
restore fnode    0C?  Y <CR>
restored fnode number:   0C
```

Fnodes 09 through 0C in the R?SAVE file have been copied into fnode 09 through 0C in the fnode file. Since the disk has not been modified since the last fnode backup, restoring the damaged fnodes should now enable you to recover the data on the disk.

EXAMPLE 2

Assume the same initial conditions as Example 1 with the following exception: two files have been modified since the last time the fnodes were backed up. In the fnode file, the new files are represented by fnodes 0D and 0E. Again, you suspect that the fnode file has been damaged, so you use the Disk Verification Utility to check the condition of data on the disk:

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
*verify
     .
     .
     .
```

After using the Disk Verification Utility to examine the structure of the disk, you find that fnodes 9 through 10 have probably been destroyed. You decide to use the RESTOREFNODE command to recover these fnodes. You do not wish to restore fnodes 0D and 0E because they were not backed up. Since the data fields of fnodes 0D and 0E in R?SAVE contain all zeros, you would be destroying possibly useful data in the corresponding fnodes. You then use RESTOREFNODE to restore a range of fnodes that includes 0D and 0E. However, you intend to pass over the restoration of these two fnodes by responding to the confirmation prompt with some character other than "Y."

```
*restorefnode 9,10 <CR>    or   rf 9,10 <CR>
restore fnode     9?  Y <CR>
restored fnode number:    9
restore fnode    0A?  Y <CR>
restored fnode number:   0A
restore fnode    0B?  Y <CR>
restored fnode number:   0B
restore fnode    0C?  Y <CR>
restored fnode number:   0C
allocation bit not set for saved fnode
restore fnode    0D?   <CR>
allocation bit not set for saved fnode
restore fnode    0E?  n <CR>
restore fnode    0F?  Y <CR>
restored fnode number:   0F
restore fnode    10?  Y <CR>
restored fnode number:   10
```

Notice that because fnodes 0D and 0E were not allocated when they were backed up, those fnodes in R?SAVE are unallocated. Therefore, the Disk Verification Utility returns the "allocation bit not set for saved fnode" message. Since you do not wish to restore this fnode, you respond to the confirmation prompt with a character other than "Y."

The R?SAVE fnodes 09 through 0C and fnodes 0F through 10 have been copied over the corresponding fnodes in the fnode file. Fnodes 0D and 0E were not restored.

3.2.5 Restoring the Volume Label

DESCRIPTION

To restore the volume label, you must have previously reserved the backup file R?SAVE when you formatted the volume. If the volume contains the R?SAVE file, a backup copy of the volume label already exists. The FORMAT command automatically places a copy of the volume label into R?SAVE when the file is created. Thereafter, the contents of the volume label do not change and you can restore the label without fear of destroying data in the existing label.

To restore the volume label, you must invoke the Disk Verification Utility using the logical device name of the appropriate volume. If the volume label is corrupted, the Disk Verification Utility attempts to use the backup copy of the volume label in R?SAVE. When the backup label is used, the Disk Verification Utility issues a message that reads "duplicate volume label used." If this message appears when the Disk Verification Utility is activated, then the volume label is damaged. To restore the volume label, enter the Disk Verification Utility RESTOREVOLUMELABEL command. The current volume label will be overwritten with the volume label copy from R?SAVE.

PROCEDURE

1. Invoke the Disk Verification Utility, using the logical device name of the volume to be backed up.
2. If the "duplicate volume label used" message appears, the volume label must be restored. Enter the Disk Verification Utility RESTOREVOLUMELABEL command.
3. When the volume label has been restored, the Disk Verification Utility returns the message "volume label restored."

EXAMPLE

Assume that a disk drive is attached as logical device :sd:. The volume :sd: contains the R?SAVE fnode backup file. When you attempt to access files on :sd:, the system returns an E$ILLEGAL_VOLUME message. You suspect that the volume label may be damaged. You decide to check your suspicions using the Disk Verification Utility.

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
duplicate volume label used
*
```

The "duplicate volume label used" message confirms that the volume label has been damaged. You restore the volume label using the RESTOREVOLUMELABEL command.

```
*restorevolumelabel <CR>    or   rvl <CR>
volume label restored
*
```

The original volume label has been overwritten with the duplicate copy from the R?SAVE file. Attempts to access files on volume :sd: should now be successful.

3.2.6 Displaying R?SAVE Fnodes

DESCRIPTION

Any fnode (both allocated and unallocated) in the R?SAVE file can be examined by using the Disk Verification Utility DISPLAYSAVEFNODE command. The Disk Verification Utility will display vital information about the fnode (total blocks, total size, block pointers, parent node, etc.). The fnode is displayed in the same format used by the DISPLAYFNODE command.

To display an R?SAVE fnode, enter the DISPLAYSAVEFNODE command and specify the hexadecimal number of the fnode to be displayed.

PROCEDURE

1. Invoke the Disk Verification Utility using the logical device name of the appropriate volume.
2. When you receive the Disk Verification Utility prompt (*), enter the Disk Verification Utility DISPLAYSAVEFNODE command. Specify the hexadecimal number of the fnode to be displayed.
3. The Disk Verification Utility will return with an fnode display.

EXAMPLE

Assume that you cannot access a file on a disk attached as :sd:. You suspect that the fnode file may be damaged. By entering the Disk Verification Utility and displaying the file's directory, you find that the file you were unable to access is represented by fnode 3C8. You use DISPLAYFNODE to display fnode 3C8, but you are not confident of the data you see. Since the fnode for the file has been backed up since the file was last modified, you decide to use data in the R?SAVE fnode to examine the fnode file. The following command displays the data for fnode 3C8 in R?SAVE.

```
super- diskverify :sd: <CR>
iRMX II Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
:sd:, outstanding connections to device have been deleted
        .
        .
        .

*displaysavefnode 3C8 <CR>    or   dsf 3C8 <CR>


    Fnode number = 3C8 (saved)
    path name: /USER/MYFILE
                      flags : 0025  =>  short file
                       type : 08  =>  data file
          file gran/vol gran : 01
                      owner : 0001
     create,access,mod times : 00000000, 00000000, 00000000
      total size,total blocks: 00002D01, 0000000C
           block pointer (1) : 000C, 004910
           block pointer (2) : 0000, 000000
           block pointer (3) : 0000, 000000
           block pointer (4) : 0000, 000000
           block pointer (5) : 0000, 000000
           block pointer (6) : 0000, 000000
           block pointer (7) : 0000, 000000
           block pointer (8) : 0000, 000000
                   this size : 00003000
                    id count : 0001
                accessor (1) : 0F, 0001
                accessor (2) : 00, 0000
                accessor (3) : 00, 0000
            parent, checksum : 03C4, 0000
                     aux (*) : 000000
*
```

You can modify the contents of the both the original fnode file and the saved fnode file by using either the EDITFNODE or EDITSAVEFNODE commands.

APPENDIX A
STRUCTURE OF A NAMED VOLUME

A.1 INTRODUCTION

This appendix describes the structure of an iRMX II volume that contains named files. It is provided as reference information to help you interpret output from the DISKVERIFY commands or to help you create your own formatting utility programs.

This appendix is for programmers with experience in reading and writing actual volume information. It does not attempt to teach these functions.

A.2 VOLUME STRUCTURE

This appendix discusses the structure of named file volumes in detail. It covers the structure of directory files and the concepts of long and short files. It also includes information on

- ISO Volume Label
- iRMX Volume Label
- fnode file
- volume free space map file
- free fnodes map file
- bad blocks map file
- root directory

The blocks reserved for the Bootstrap Loader (Figure A-1) are not discussed. Bootstrap Loader blocks are automatically included on a new volume when you format a volume with the FORMAT command. Refer to the FORMAT command for a description of the bootstrap option.

Figure A-1 illustrates the general structure of a named file volume.

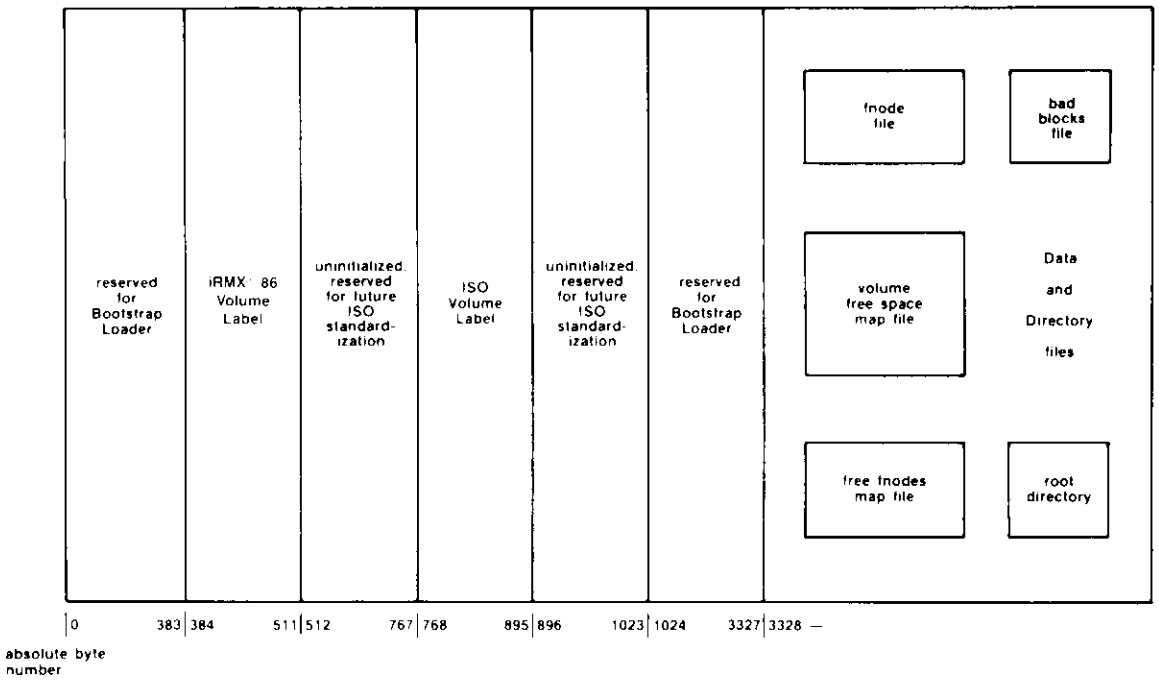


Figure A-1. General Structure of Named Volumes

A.3 VOLUME LABELS

Each iRMX II named volume contains ISO (International Standardization Organization) label information as well as iRMX II label information and files. This section describes the structure of ISO volume labels and iRMX II volume labels, both of which must be present on a named volume.

A.3.1 ISO Volume Label

The ISO volume label is recorded in absolute byte positions 768 through 895 of the volume (for example, sector 07 of a single-density flexible diskette). The structure of this volume label (in PL/M notation) is:

```
DECLARE
    ISO$VOL$LABEL   STRUCTURE(
                        LABEL$ID(3)         BYTE,
                        RESERVED$A          BYTE,
                        VOL$NAME(6)         BYTE,
                        VOL$STRUC           BYTE,
                        RESERVED$B(60)      BYTE,
                        REC$SIDE            BYTE,
                        RESERVED$C(4)       BYTE,
                        ILEAVE(2)           BYTE,
                        RESERVED$D          BYTE,
                        ISO$VERSION         BYTE,
                        RESERVED$E(48)      BYTE);
```

Where:

LABEL$ID(3)	Label identifier. For named file volumes, this field contains the ASCII characters "VOL".
RESERVED$A	Reserved field containing the ASCII character "1".
VOL$NAME(6)	Volume name. This field can contain up to six printable ASCII characters, left justified and space filled. A value of all spaces implies that the volume name is recorded in the iRMX II Volume Label (absolute byte positions 384-393).
VOL$STRUC	For named file volumes, this field contains the ASCII character "N", indicating that this volume has a non-ISO file structure.
RESERVED$B(60)	Reserved field containing 60 bytes of ASCII spaces.
REC$SIDE	For named file volumes, this field contains the ASCII character "1" to indicate that only one side of the volume is to be recorded.
RESERVED$C(4)	Reserved field containing four bytes of ASCII spaces.
ILEAVE(2)	Two ASCII digits indicating the interleave factor for the volume, in decimal. ASCII digits consist of the numbers 0 through 9. When formatting named volumes, you should set this field to the same interleave factor that you use when physically formatting the volume.
RESERVED$D	Reserved field containing an ASCII space.

ISO$VERSION	For named file volumes, this field contains the ASCII character "1", which indicates ISO version number one.
RESERVED$E(48)	Reserved field containing 48 ASCII spaces.

A.3.2 iRMX® II Volume Label

The iRMX II Volume Label is recorded in absolute byte positions 384 through 511 of the volume (sector 04 of a single density flexible diskette). The structure of this volume label is as follows:

```
DECLARE
    RMX$VOLUME$INFORMATION  STRUCTURE(
                                VOL$NAME(10)        BYTE,
                                FLAGS               BYTE,
                                FILE$DRIVER         BYTE,
                                VOL$GRAN            WORD,
                                VOL$SIZE            DWORD,
                                MAX$FNODE           WORD,
                                FNODE$START         DWORD,
                                FNODE$SIZE          WORD,
                                ROOT$FNODE          WORD,
                                DEV$GRAN            WORD,
                                INTERLEAVE          WORD,
                                TRACK$SKEW          WORD,
                                SYSTEM$ID           WORD,
                                SYSTEM$NAME(12  )   BYTE,
                                DEVICE$SPECIAL(8)   BYTE);
```

where:

VOL$NAME(10)	Volume name in printable ASCII characters, left justified and zero filled.
FLAGS	BYTE that lists the device characteristics for automatic device recognition. The individual bits in this BYTE indicate the following characteristics (bit 0 is rightmost bit):

Bit	Meaning
0	VF$AUTO flag. When set to one, this bit indicates that the FLAGS byte contains valid data for automatic device recognition. When set to zero, it indicates that the remaining flags contain meaningless data.

	1	VF$DENSITY flag. This bit indicates the recording density of the volume. When set to one, it indicates modified frequency modulation (MFM) or double-density recording. When set to zero, it indicates frequency modulation (FM) or single-density recording.
	2	VF$SIDES flag. This bit indicates the number of recording sides on the volume. When set to one, it indicates a double-sided volume. When set to zero, it indicates a single-sided volume.
	3	VF$MINI flag. This bit indicates the size of the recording media. When set to one, it indicates a 5 1/4-inch volume. When set to zero, it indicates an 8-inch volume.
	4	VFNOTFLOPPY. This bit indicates the type of disk you are using. When this and Bit 0 are set to 1, it indicates a Winchester disk.
	5-7	Reserved
FILE$DRIVER	Number of the file driver used with this volume. For named file volumes, this field is set to four.	
VOL$GRAN	Volume granularity, specified in bytes. This value must be a multiple of the device granularity. It sets the size of a logical device block, also called a volume block.	
VOL$SIZE	Size of the entire volume, in bytes.	
MAX$FNODE	Number of fnodes in the fnode file. (Refer to the next section for a description of fnodes.)	
FNODE$START	A 32-bit value that represents the number of the first byte in the fnode file (byte 0 is the first byte of the volume).	
FNODE$SIZE	Size of an fnode, in bytes.	
ROOT$FNODE	Number of the fnode describing the root directory. (Refer to the next section for further information.)	
DEV$GRAN	Device granularity of all tracks except track zero (which contains the volume label). This field is important only when the system requires automatic device recognition.	
INTERLEAVE	Block interleave factor for this volume. This value indicates the physical distance, in blocks, between consecutively-numbered blocks on the volume. A value of one indicates that consecutively-numbered blocks are adjacent. A value of zero indicates an unknown or undefined interleave factor.	

TRACK$SKEW — Offset, in bytes, between the first block on one track and the first block on the next track. A value of zero indicates that all tracks are identical.

SYSTEM$ID — Numerical code identifying the operating system that formatted the volume. The following codes are reserved for Intel operating systems:

Operating System	Code
iRMX I, II	0 - 0Fh
iRMX 88	10h - 1Fh
iNDX	20h - 2Fh

Currently, the iRMX II Operating System places a zero in this field.

SYSTEM$NAME(12) — Name of the operating system that formatted the volume, in printable ASCII characters, left justified and space filled. Zeros (ASCII nulls) indicate that the operating system is unknown. The iRMX II Operating System currently places several pieces of information into this field, as follows:

- The leftmost eight bytes of this field contain the ASCII characters "iRMX II.3" to identify the operating system. The iRMX II.1 Operating System filled this field with zeros.
- The next byte is an ASCII character that identifies the program that formatted the volume. The following characters apply:

Character	Formatting Program
F	Human Interface FORMAT command
U	iRMX I Files Utility (used prior to iRMX I.7)

If the formatting program is unable to provide this information, it places an ASCII space in this field.

- The Human Interface FORMAT command that is part of iRMX II.3 places the characters "03 " in the last 3 bytes of this field.

DEVICE$SPECIAL(8) Reserved for special device-specific information. When no device-specific information exists, this field must contain zeros. For example, if the device is a Winchester disk with an iSBC 214/215G controller, the iRMX II Operating System imposes a structure on this field and supplies the following information:

```
SPECIAL STRUCTURE(
            CYLINDERS     WORD,
            FIXED         BYTE,
            REMOVABLE     BYTE,
            SECTORS       BYTE,
            SECTOR SIZE   WORD,
            ALTERNATES    BYTE);
```

where:

CYLINDERS	Total number of cylinders on the disk drive.
FIXED	Number of heads on the fixed disk or Winchester disk.
REMOVABLE	Number of heads on the removable disk cartridge.
SECTORS	Number of sectors in a track.
SECTOR SIZE	Sector size, in bytes.
ALTERNATES	Number of alternate cylinders or spare sectors on a track.

The remainder of the Volume Label (bytes 440 through 511) is reserved and must be set to zero.

A.4 INITIAL FILES

Any mechanism that formats iRMX II named volumes must place seven files, with the option of an eighth file, on the volume during the format process. These files are

File	File Name
fnode file	
volume label file	R?VOLUMELABEL
volume free space map file	R?SPACEMAP
free fnodes map file	R?FNODEMAP
bad blocks file	R?BADBLOCKMAP
root directory	
space accounting file,	
Optionally, duplicate volume label file	R?SAVE

The first of these files, the fnode file, contains information about all of the files on the volume. The general structure of the fnode file is discussed first. Then all of the files are discussed in terms of their fnode entries and their functions.

A.4.1 Fnode File

A data structure called a file descriptor node (fnode) describes each file in a named file volume. All the fnodes for the entire volume are grouped together in a file called the fnode file. When the I/O System accesses a file on a named volume, it examines the iRMX II Volume Label (described in the previous section) to determine the location of the fnode file, and then examines the appropriate fnode to determine the actual location of the file.

When a volume is formatted, the fnode file contains seven allocated fnodes and any number of unallocated fnodes. The original number of unallocated fnodes depends on the FILES parameter of the FORMAT command. These allocated fnodes represent the fnode file, the volume label file, the volume free space map file, the free fnodes map file, the bad blocks file, the root directory, and the space accounting file. (Later sections of this appendix describe these files.) The size of the fnode file is determined by the number of fnodes that it contains. The number of fnodes in the fnode file also determines the number of files that can be created on the volume. The number of files is set when you format the storage medium.

The structure of an individual fnode in a named file volume is as follows:

```
DECLARE
    FNODE STRUCTURE(
            FLAGS    WORD,
            TYPE     BYTE,
            GRAN     BYTE,
            OWNER    WORD,
            CR$TIME  DWORD,
            ACCESS$TIME DWORD,
            MOD$TIME    DWORD,
            TOTAL$SIZE  DWORD,
            TOTAL$BLKS  DWORD,

            POINTR(40)  BYTE,
            THIS$SIZE   DWORD,
            RESERVED$A  WORD,
            RESERVED$B  WORD,
            ID$COUNT    WORD,

            ACC(9)  BYTE,
            PARENT  WORD,
            AUX(*)  BYTE);
```

where:

FLAGS — A WORD that defines a set of attributes for the file. The individual bits in this word indicate the following attributes (bit 0 is the rightmost bit):

Bit	Meaning
0	Allocation status. If set to one, this fnode describes an actual file. If set to zero, this fnode is available for allocation. When formatting a volume, this bit is set to one in the six allocated fnodes. In other fnodes, it is set to zero.

	1	Long or short file attribute. This bit describes how the PTR fields of the fnode are interpreted. If set to zero, indicating a short file, the PTR fields identify the actual data blocks of the file. If set to one, indicating a long file, the PTR fields identify indirect blocks (described later in this section). When formatting a volume, this bit is always set to zero, since the initial files on the volume are short files.
	2	Reserved bit, always set to one.
	3-4	Reserved bits, always set to zero.
	5	Modification attribute. Whenever a file is modified, this bit is set to one. Initially, when a volume is formatted, this bit is set to zero in each fnode.
	6	Deletion attribute. This bit is set to one to indicate that the file is a temporary file or that the file will be deleted (the deletion may be postponed because additional connections exist to the file). Initially, when the volume is formatted, this bit is set to zero in each fnode.
	7-15	Reserved bits, always set to zero.

TYPE — Type of file. The following are acceptable types:

Mnemonic	Value	Type
FT$FNODE	0	fnode file
FT$VOLMAP	1	volume free space map
FT$FNODEMAP	2	free fnodes map
FT$ACCOUNT	3	space accounting file
FT$BADBLOCK	4	device bad blocks file
FT$DIR	6	directory file
FT$DATA	8	data file
FT$VLABEL	9	volume label file

During system operation, only the I/O System can access file types other than FT$DATA and FT$DIR. These file types are discussed later in this section.

GRAN — File granularity, specified in multiples of the volume granularity. The default value is 1. This value can be set to any multiple of the volume granularity.

OWNER	User ID of the owner of the file. For the files initially present on the volume, this parameter is important only for the root directory. For the root directory, this parameter should specify the user WORLD (FFFFH). The I/O System does not examine this parameter for the other files (fnode file, volume free space map file, free fnodes map file, bad blocks file, volume label), so a value of zero can be specified.
CR$TIME	Time and date that the file was created, expressed as a 32-bit value. This value indicates the number of seconds since a fixed, user-determined point in time. By convention, this point in time is 12:00, January 1, 1978. For the files initially present on the volume, this parameter is important only for the root directory. A zero can be specified for the other files (fnode file, volume free space map file, free fnodes map file, bad blocks file, volume label.)
ACCESS$TIME	Time and date of the last file access (read or write), expressed as a 32-bit value. For the files initially present on the volume, this parameter is important only for the root directory.
MOD$TIME	Time and date of the last file modification, expressed as a 32-bit value. For the files initially present on the volume, this parameter is important only for the root directory.
TOTAL$SIZE	Total size, in bytes, of the actual data in the file.
TOTAL$BLKS	Total number of volume blocks used by this file, including indirect block overhead. A volume block is a block of data whose size is the same as the volume granularity. All memory in the volume is divided into volume blocks, which are numbered sequentially, starting with the block containing the smallest addresses (block 0). Indirect blocks are discussed later in this section.
POINTR(40)	A group of BYTES on which the following structure is imposed:

```
PTR(8) STRUCTURE(
           NUM$BLOCKS  WORD,
           BLK$PTR(3)  BYTE);
```

This structure identifies the data blocks of the file. These data blocks may be scattered throughout the volume, but together they make up a complete file. If the file is a short file (bit 1 of the FLAGS field is set to zero), each PTR structure identifies an actual data block. In this case, the fields of the PTR structure contain the following:

NUM$BLOCKS	Number of volume blocks in the data block.
BLK$PTR(3)	A 24-bit value specifying the number of the first volume block in the data block. Volume blocks are numbered sequentially, starting with the block with the smallest address (block 0). The bytes in the BLK$PTR array range from least significant (BLK$PTR(0)) to most significant (BLK$PTR(2)).

If the file is a long file (bit 1 of the FLAGS field is set to one), each PTR structure identifies an indirect block (possibly consisting of more than one contiguous volume block), which in turn identifies the data blocks of the file. In this case, the fields of the PTR structure contain the following:

NUM$BLOCKS	Number of volume blocks pointed to by the indirect block.
BLK$PTR(3)	A 24-bit volume block number of the indirect block.

Indirect blocks are discussed later in this section.

THIS$SIZE	Size, in BYTES, of the total data space allocated to the file. This figure does not include space used for indirect blocks, but it does include any data space allocated to the file, regardless of whether the file fills that allocated space.
RESERVED$A	Reserved field, set to zero.
RESERVED$B	Reserved field, set to zero.
ID$COUNT	Number of access-ID pairs declared in the ACC(9) field.
ACC(9)	A group of BYTES on which the following structure is imposed:

```
ACCESSOR(3) STRUCTURE(
                ACCESS    BYTE,
                ID        WORD);
```

This structure contains the access-ID pairs that define the access rights for the users of the file. By convention, when a file is created, the owner's ID is inserted in ACCESSOR(0), along with the code for the access rights. The fields of the ACCESSOR structure contain the following:

ACCESS — Encoded access rights for the file. The settings of the individual bits in this field grant (if set to one) or deny (if set to zero) permission for the corresponding operation. Bit 0 is the rightmost bit.

Bit	Data File Operation	Directory Operation
0	delete	delete
1	read	list
2	append	add entry
3	update	change entry
4-7	reserved (must be 0)	

ID — ID of the user who gains the corresponding access permission.

PARENT — Fnode number of directory file that lists this file. For files initially present on the volume, this parameter is important only for the root directory. For the root directory, this parameter should specify the number of the root directory's own fnode. For other files (fnode file, volume free space map file, free fnodes map file, bad blocks file, volume label) the I/O System does not examine this field.

AUX(*) — Auxiliary BYTES associated with the file. The named file driver does not interpret this field, but the user can access it by making GET$EXTENSION$DATA and SET$EXTENSION$DATA system calls. The size of this field is determined by the size of the fnode, specified in the iRMX II Volume Label. If you use the Human Interface FORMAT command or create your own utility to format a volume, you can make this field as large as you wish; however, a larger AUX field implies slower file access.

Certain fnodes designate special files that appear on the volume. The following sections discuss these fnodes and the associated files.

A.4.2 Fnode 0 (Fnode File)

The first fnode structure in the fnode file describes the fnode file itself. This file contains all the fnode structures for the entire volume. It must reside in contiguous locations in the volume. The fields of fnode 0 must be set as follows:

- The bits in the FLAGS field are set to the following (bit 0 is the rightmost bit):

Bit	Value	Description
0	1	Allocated file
1	0	Short file
2	1	Primary fnode
3-4	0	Reserved bits
5	0	Initial status is unmodified
6	0	File will not be deleted
7-15	0	Reserved bits

- The TYPE field is set to FT$FNODE.
- The GRAN field is set to 1.
- The OWNER field is set to the ID of the user who formatted it.
- The CR$TIME, ACCESS$TIME, and MOD$TIME fields are set to the time the system was formatted.
- Since the iRMX II Volume Label specifies the size of an individual fnode structure and the number of fnodes in the fnode file, the value specified in the TOTAL$SIZE field of fnode 0 must equal the product of the values in the FNODE$SIZE and MAX$FNODE fields of the iRMX II Volume Label.
- The TOTAL$BLOCKS field specifies enough volume blocks to account for the memory listed in the TOTAL$SIZE field. The product of the value in the TOTAL$BLOCKS field and the volume granularity equals the value of the THIS$SIZE field, since the fnode file is a short file.
- Since the fnode file must reside in contiguous locations in the volume, only one PTR structure describes the location of the file. The value in the NUM$BLOCKS field of that PTR structure equals the value in the TOTAL$BLOCKS field. The BLK$PTR field indicates the number of the first block of the fnode file.
- The ID$COUNT field is set to one.

A.4.3 Fnode 1 (Volume Free Space Map File)

The second fnode, fnode 1, describes the volume free space map file. The TYPE field for fnode 1 is set to FT$VOLMAP to designate the file as such.

The volume free space map file keeps track of all the space on the volume. It is a bit map of the volume, in which each bit represents one volume block (a block of space whose size is the same as the volume granularity). If a bit in the map is set to one, the corresponding volume block is free to be allocated to any file. If a bit in the map is set to zero, the corresponding volume block is already allocated to a file. The bits of the map correspond to volume blocks such that bit n of byte m represents volume block (8 * m) + n. The bits in the remaining space allocated to the map file (those that do not correspond to actual blocks of memory) must be set to zero.

When the volume is formatted, the volume free space map file indicates that the first 3328 bytes of the volume (the label and bootstrap information) plus any files initially placed on the volume (fnode file, volume free space map file, free fnodes map file, bad blocks file) are allocated.

A.4.4 Fnode 2 (Free Fnodes Map File)

The third fnode, fnode 2, describes the free fnodes map file. The TYPE field of fnode 2 is set to FT$FNODEMAP to designate the file as such.

The free fnodes map file keeps track of all the fnodes in the fnodes file. It is a bit map in which each bit represents an fnode. If a bit in the map is set to one, the corresponding fnode is not in use and does not represent an actual file. If a bit in the map is set to zero, the corresponding fnode already describes an existing file. The bits in the map correspond to fnodes such that bit n of byte m represents fnode number (8 * m) + n. The bits in the remaining space allocated to the map file (those that do not correspond to actual fnode structures) must be set to zero.

When the volume is formatted, the free fnodes map file indicates that fnodes 0, 1, 2, 3, 4, 5, and 6 are in use. If other files are initially placed on the volume, the free fnodes map file must be set to indicate this as well.

A.4.5 Fnode 3 (Accounting File)

When a volume is formatted, fnode 3 is set up representing a file of type FT$ACCOUNT. The fnode is set up as allocated, and of the indicated type, but it does not assign any actual space for the file.

A.4.6 Fnode 4 (Bad Blocks Map File)

The fifth fnode, fnode 4, contains a map of all the bad blocks on the volume. The TYPE field of fnode 4 is set to FT$BADBLOCK to indicate this.

The bad block map file keeps track of all the bad blocks on the volume. It is a bit map of the volume, in which each bit represents one volume block (a block of space whose size is the same as the volume granularity). If a bit in the map is set to zero, the corresponding volume block has no bad blocks and may be allocated to any file. If a bit in the map is set to one, the corresponding volume block is bad. If a block is marked "bad", it must also be marked allocated in the volume free space file. The bits of the map correspond to volume blocks such that bit n of byte m represents volume block (8 * m) + n.

A.4.7 Fnode 5 (Volume Label File)

This fnode contains the first 3328 bytes of any volume. The information in this file defines the volume as a whole. The TYPE field of this fnode is set to FT$VLABEL. You cannot write to this fnode.

A.4.8 Fnode 6 (Root Directory)

The root directory is a special directory file. It is the root of the named file hierarchy for the volume. The iRMX II Volume Label specifies the fnode number of the root directory. The root directory is its own parent. That is, the PARENT field of its fnode specifies its own fnode number.

The root directory (and all directory files) associates file names with fnode numbers. It consists of a number of entries that have the following structure:

```
DECLARE
    DIR$ENTRY   STRUCTURE(
                    FNODE           WORD,
                    COMPONENT(14)   BYTE);
```

where:

FNODE	Fnode number of a file listed in the directory.
COMPONENT(14)	A string of ASCII characters that is the final component of the path name identifying the file. This string is left justified and null padded to 14 characters.

When a file is deleted, its fnode number in the directory entry is set to zero.

A.4.9 Fnode 7 (R?SAVE)

The R?SAVE is a file which may be optionally created by the RESERVE option of the FORMAT command. The FORMAT command creates a file named R?SAVE, which contains the duplicate volume label, in the innermost track of the volume. A copy of the iRMX II volume label is placed in the front (that is, the physical end) of the file and an fnode is allocated for R?SAVE in the fnode file. (The fnode for the R?SAVE file is allocated out of the fnodes reserved through the FILES parameter of the FORMAT command.)

The FORMAT command creates a backup of the fnode file in its initialized state. R?SAVE is not subsequently updated as files are written to or deleted from the volume. Therefore, you will have to use the BACKUPFNODES Disk Verification Utility command or the BACKUP option of the Human Interface SHUTDOWN command to back up the fnode file at regular intervals.

A.4.10 Other Fnodes

When formatting a volume, no other fnodes in the fnode file represent actual files. The remaining fnodes must have bit zero (allocation status) set to zero.

A.5 LONG AND SHORT FILES

A file on a volume is not necessarily one contiguous string of bytes. In many cases, it consists of several contiguous blocks of data scattered throughout the volume. The fnode for the file indicates the locations and sizes of these blocks in one of two ways, as short files or as long files.

A.5.1 Short Files

If the file consists of eight or less distinct blocks of data, its fnode can specify it as a short file. The fnode for a short file has bit 1 of the FLAGS field set to zero. This indicates to the I/O System that the PTR structures of the fnode identify the actual data blocks that make up the file. Figure A-2 illustrates an fnode for a short file. Decimal numbers are used in the figure for clarity.

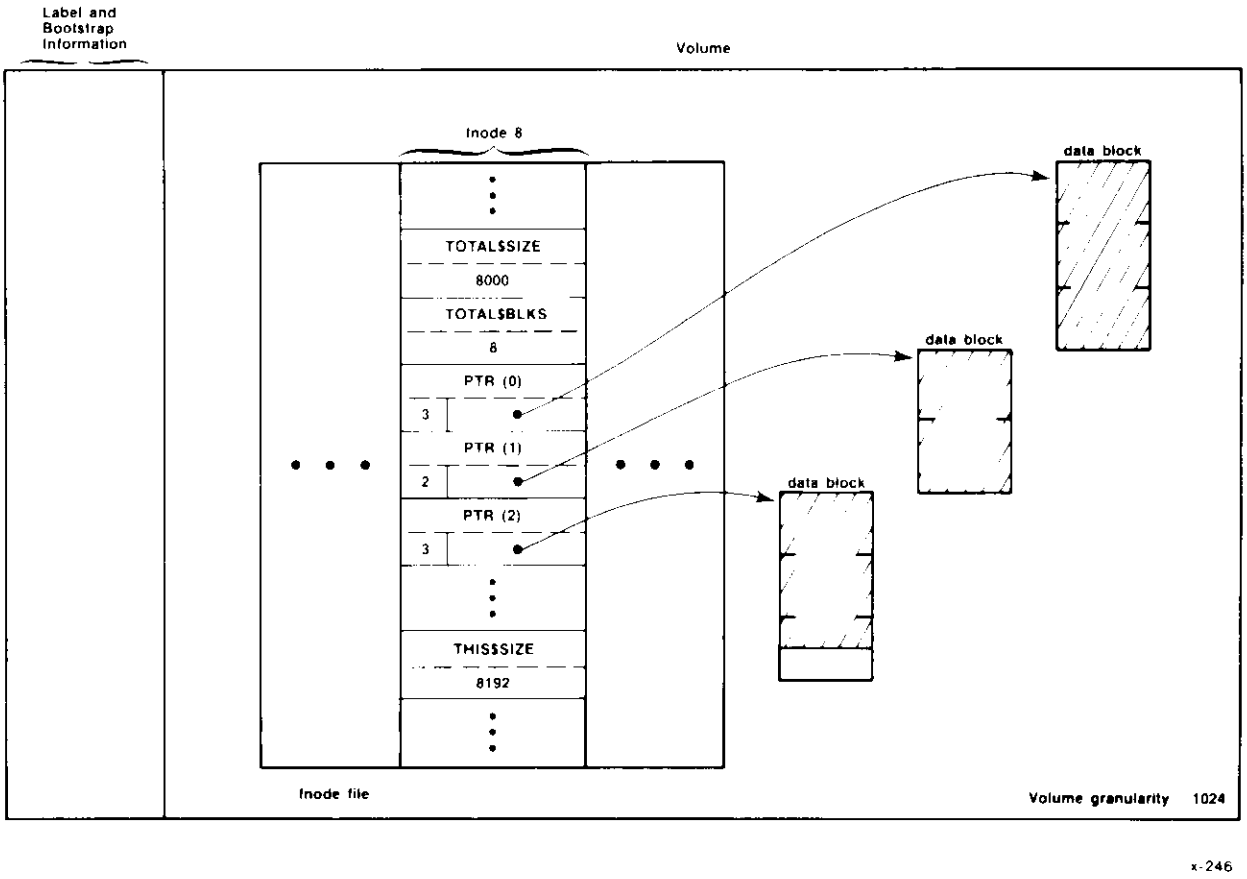


Figure A-2. Short File Fnode

As you can see in Figure A-2, fnode 8 identifies the short file. The file consists of three distinct data blocks. Three PTR structures give the locations of the data blocks. The NUM$BLOCKS field of each PTR structure gives the length of the data block (in volume blocks), and the BLK$PTR field points to the first volume block of the data block.

The other fields shown in Figure A-2 include TOTAL$BLKS, THIS$SIZE, and TOTAL$SIZE. The TOTAL$BLKS field specifies the number of volume blocks allocated to the file, which in this case is eight. This equals the sum of NUM$BLOCKS values (3 + 2 + 3), since short files use all allocated space as data space.

The THIS$SIZE field specifies the number of bytes of data space allocated to the file. This is the sum of the NUM$BLOCKS values (3 + 2 + 3) multiplied by the volume granularity (1024) and equals 8192.

The TOTAL$SIZE field specifies the number of bytes of data space that the file occupies (designated in Figure A-2 by the shaded area). As you can see, the file does not occupy all the space allocated for it, so the TOTAL$SIZE value (8000) is not as large as the THIS$SIZE value.

A.5.2 Long Files

If the file consists of more than eight distinct blocks of data, its fnode must specify it as a long file. The fnode for a long file has bit 1 of the FLAGS field set to one. This tells the I/O System that the PTR structures of the fnode identify indirect blocks. The indirect blocks identify the actual data blocks that make up the file.

Each indirect block contains a number of indirect pointers, which are structures similar to the PTR structures. However, an indirect block can contain more than eight structures and thus can point to more than eight data blocks. In fact, an indirect block can consist of more than one volume block; however, all volume blocks of an indirect block must be contiguous. The structure of each indirect pointer is as follows:

```
DECLARE
    IND$PTR STRUCTURE(
                NBLOCKS    BYTE,
                BLK$PTR    BLOCK$NUM);
```

where:

NBLOCKS	Number of volume blocks in the data block.
BLK$PTR	A 24-bit volume block number of the first volume block in the data block. Volume blocks are numbered sequentially throughout the volume, starting with the block with the smallest address (block 0).

The Operating System determines how many indirect pointers there are in an indirect block by comparing the NBLOCKS fields of the indirect pointers with the NUM$BLOCKS field of the fnode. It assumes that the indirect block contains as many pointers as necessary for the sum of the NBLOCKS fields to equal the NUM$BLOCKS field.

Figure A-3 illustrates an fnode for a long file. Decimal numbers are used in the figure for clarity.

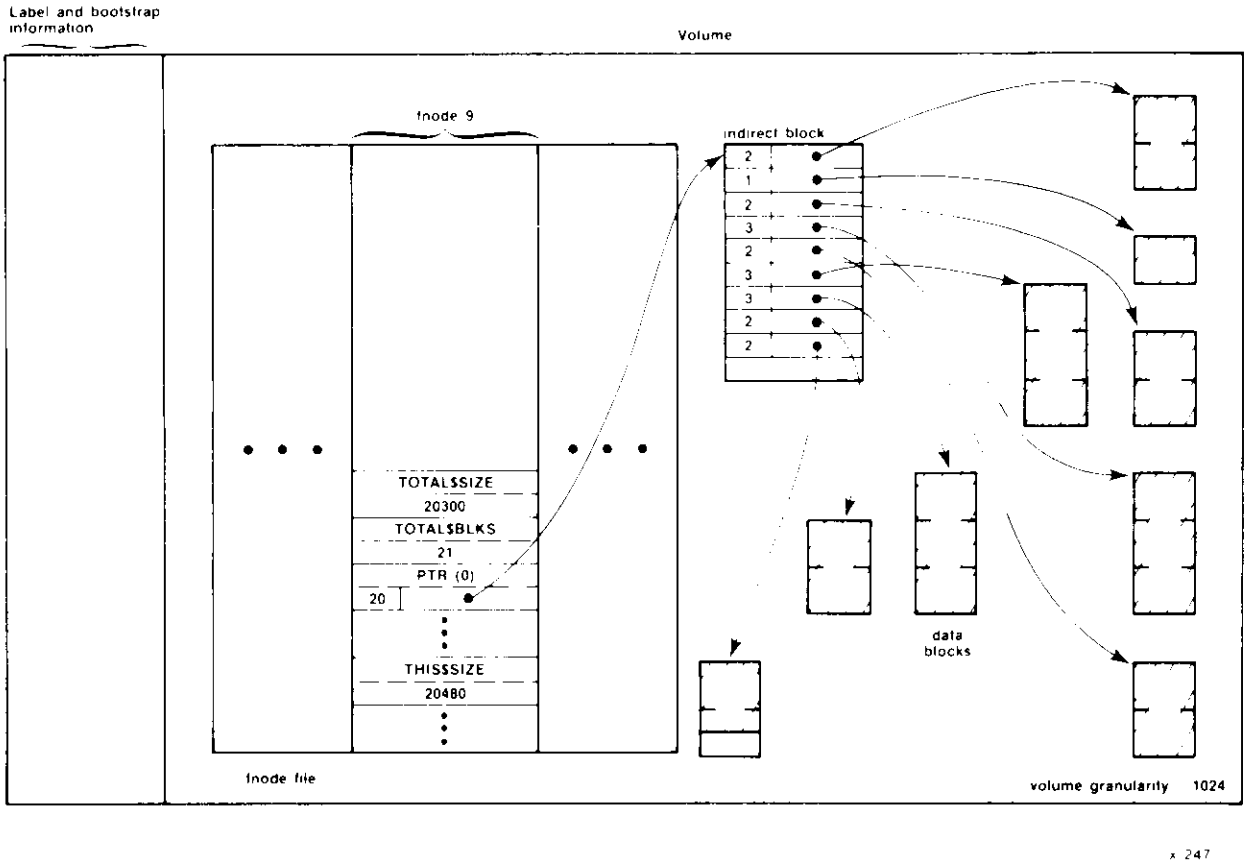


Figure A-3. Long File FNODE

As you can see in Figure A-3, fnode 9 identifies the long file. The actual file consists of nine distinct data blocks. One PTR structure and an indirect block give the locations of the data blocks. The NUM$BLOCKS field of the PTR structure contains the number of volume blocks pointed to by the indirect block. The BLK$PTR field points to the first volume block of the indirect block.

In the indirect block, each NBLOCKS field gives the length of an individual data block, and each BLK$PTR field points to the first volume block of a data block.

Figure A-3 also lists the TOTAL$BLKS, THIS$SIZE, and TOTAL$SIZE values, which are more complex than for a short file. The TOTAL$BLKS field specifies the number of volume blocks allocated to the file, which in this case is 21. Of these 21, 20 are used for actual data storage and 1 is used for the indirect block.

The THIS$SIZE field specifies the number of bytes of data space allocated to the file, and does not include the size of the indirect block. This size is equal to the NUM$BLOCKS value (20) or the sum of NBLOCKS values in the indirect block (2 + 1 + 2 + 3 + 2 + 3 + 3 + 2 + 2 = 20) multiplied by the volume granularity (1024) and equals 20480.

The TOTAL$SIZE field specifies the number of bytes of data space that the file currently occupies (designated in Figure A-3 by the shaded areas). As you can see, the file does not occupy all the space allocated for it, so the TOTAL$SIZE value (20300) is not as large as the THIS$SIZE value.

A.6 FLEXIBLE DISKETTE FORMATS

The flexible diskette device drivers supplied with the iRMX II Basic I/O System can support several diskette characteristics, listed in Tables A-1 and A-2.

Table A-1. 8-Inch Diskette Characteristics

Sector Size	Density	Sectors per Track	Device Size (in bytes) One-Sided	Two-Sided
128	Single	26	256256	512512
256	Single	15	295168	590848
512	Single	8	314880	630272
1024	Single	4	315392	630784
256	Double	26	509184	1021696
512	Double	15	587264	1177600
1024	Double	8	626688	1255424

Table A-2. 5 1/4-Inch Diskette Characteristics

Sector Size	Density	Sectors per Track	Device Size (in bytes) One-Sided 40 Tracks	One-Sided 80 Tracks	Two-Sided 40 Tracks	Two-Sided 80 Tracks
128	Single	16	81920	163840	163840	327680
256	Single	9	91904	184064	184064	368384
512	Single	4	81920	163840	163840	327680
1024	Single	2	81920	163840	163840	327680
256	Double	16	1617921	325632	325632	653312
512	Double	8	1617921	325632	325632	653312
1024	Double	4	1617921	325632	325632	653312

For compatibility with ECMA (European Computer Manufacturers Association) and ISO (International Organization for Standardization), the iRMX II device drivers, when called by the Human Interface FORMAT command, can format the beginning tracks of all flexible diskettes in the same way. A configuration option for each driver enables you to specify the following:

- For all 5 1/4-inch and 8-inch flexible diskettes, the device drivers format track 0 of side 0 with single-density, 128-byte sectors, with an interleave factor of 1.
- For 8-inch, double-sided, double-density flexible diskettes, the device drivers format track 0 of side 1 with double-density, 256-byte sectors.

The iRMX II device drivers map the sectors on these beginning tracks into blocks of device granularity size so that the Basic I/O System and the Bootstrap Loader can treat flexible diskettes as if they contained a contiguous string of blocks, all of the same size.

However, this mapping is not exact when you use 8-inch, double-sided, double-density diskettes and specify a device granularity of 512 or 1024. A problem arises because there are 26 128-byte sectors in a track, which is not an integral mapping for device granularities of 512 or 1024. Thus, the device driver combines the leftover 128-byte sectors of track 0, side 0 with the first sectors of track 0, side 1 to make a block of device granularity size. This continues throughout track 0, side 1, but the same problem occurs with the last 256-byte sectors of track 0, side 1; not enough sectors are available to make a block of device granularity size.

When the device driver tries to combine these leftover sectors of track 0, side 1 with the first sectors of track 1, side 0, it finds that the sectors of track 1, side 0 are already of device granularity size. Therefore, since the device driver cannot access partial sectors, it is left with one block (the leftover sectors of track 0, side 1) that is less than device granularity size. When the device granularity is 512, this small block is block 19; when the device granularity is 1024, it is block 9.

If nothing is done to exclude this smaller-than-normal block from use, the device driver will treat this block as a normal block, assuming it is of device granularity size. Thus, if you try to write information to that block, the driver will attempt to write an entire device granularity block of information into a block that is much smaller, thereby losing data.

To prevent this situation, the Human Interface FORMAT command automatically declares this smaller-than-normal block as allocated in the volume free space map when it formats the volume. This prevents the Basic I/O System from ever writing information into this block. If you write your own formatting utility, you should also declare this block as allocated.

< command 2-6, 29
<CR> command 2-6
> command 2-6, 28
5 1/4-inch diskette characteristics A-22
8-inch diskette characteristics A-21

A

Aborting commands 2-4
Add command 2-46
Address command 2-46
Allocate command 2-5, 8
Argument error 2-5
Automatic device recognition A-4, 5

B

Backing up the volume label 3-6
Backupfnodes command 2-5, 11
Bad blocks 2-8, 44, 3-2
Bad blocks map file 2-9, 59, 60, 65, 66, 69, 72, , A-8, 16
Bad track information, displaying 1-3
BF command 2-5, 11
Block allocation 2-8
Block command 2-47
Block I/O error 2-4
Bootstrap Loader blocks A-1

C

Checksums 1-4, 2-31, 36, 65
Command options
    All 1-5
    Disk 1-3
    Fix 1-4
    Getbadtrackinfo 1-3
    List 1-5

Command options (cont.)
Named 1-4
Named1 1-4
Named2 1-4
Physical 1-5
Verify 1-3
Commands
< 2-6, 29
<CR> 2-6
> 2-6, 28
Aborting 2-4
Allocate 2-5
Backupfnodes 2-5
BF 2-5, 11
D 2-6, 16
DB 2-6, 16
DD 2-6, 20
DF 2-6, 23
Disk 2-6, 13
Displaybyte 2-6, 16
Displaydirectory 2-6, 20
Displayfnode 2-6, 23
Displaynextblock 2-6, 28
Displaypreviousblock 2-6, 29
Displaysavefnode 2-6, 27
Displayword 2-6, 18
DNB 2-6, 28
DPB 2-6, 29
DSF 2-6
DSF command 2-27
DW 2-6, 18
E 2-6, 34
Editfnode 2-6, 30
Editsavefnode 2-6, 33
EF 2-6, 30
Error messages 2-4
ESF 2-6, 33
Exit 2-6, 34
Fix 2-6, 35
Free 2-6, 38
GB 2-6, 41
Getbadtrackinfo 2-6, 41
H 2-7, 43

Commands (cont.)
Help 2-7, 43
LBB 2-7, 44
Listbadblocks 2-7, 44
Miscellaneous 2-7, 46
Names, entering 2-2
Parameters 2-3
Q 2-7, 52
Quit 2-7, 52
R 2-7, 53
Radices 2-3
Read 2-7, 53
Restorefnode 2-7, 54
Restorevolumelabel 2-7, 57
RF 2-7, 54
RVL 2-7, 57
S 2-7, 61
Save 2-7, 59
SB 2-7, 61
Substitutebyte 2-7, 61
Substituteword 2-7, 64
Summary 2-5
SW 2-7, 64
Syntax 2-1
V 2-7, 65
Verify 2-7, 65
W 2-7, 74
Write 2-7, 74
Conventions iv

D

D command 2-6, 16
DB command 2-6, 16
DD command 2-6, 20
Dec command 2-48
DF command 2-6, 23
Directing output 1-2
Directories, displaying 2-20
Disk command 2-6, 13
Displaybyte command 2-6, 16
Displaydirectory command 2-6, 20
Displayfnode command 2-6, 23
Displaying R?SAVE 3-11
Displaynextblock command 2-6, 28

Displayprevious block command 2-29
Displaypreviousblock command 2-6
Displaysavefnode command 2-6, 27
Displayword command 2-6, 18
Div command 2-48
DNB command 2-6, 28
DPB command 2-6, 29
DSF command 2-6, 27
Duplicate volume label file 3-2, A-8, 17
DW command 2-6, 18

E

E command 2-6, 34
Editfnode command 2-6, 30
Editsavefnode command 2-6, 33
EF command 2-6, 30
Error Messages 1-6, 2-4
- Add 2-51
- Address 2-51
- Allocate 2-10
- Backupfnodes 2-12
- BF 2-12
- Block 2-51
- D 2-17
- DB 2-17
- DD 2-21
- Dec 2-51
- DF 2-26
- Displaybyte 2-17
- Displaydirectory 2-21
- Displayfnode 2-26
- Displaysavefnode 2-27
- Div 2-51
- DSF 2-27
- Editfnode 2-31
- Editsavefnode 2-33
- EF 2-31
- ESF 2-33
- Free 2-39
- GB 2-42
- Getbadtrackinfo 2-42
- Hex 2-51

Error Messages (cont.)
LBB 2-45
Listbadblocks 2-45
Miscellaneous commands 2-51
Mod 2-51
Mul 2-51
R 2-53
Read 2-53
Restorefnode 2-55
Restorevolumelabel 2-58
RF 2-55
RVL 2-58
S 2-62
Save 2-60
SB 2-62
Sub 2-51
Substitutebyte 2-62
V 2-69
Verify 2-69
W 2-75
Write 2-75
ESF command 2-6, 33
Examples
Add 2-51
Address 2-51
Backupfnodes 2-12
BF 2-12
Block 2-51
D 2-17
DB 2-17
DD 2-22
Dec 2-51
DF 2-26
Disk 2-15
Displaybyte 2-17
Displaydirectory 2-22
Displayfnode 2-26
Displaysavefnode 3-12
Displaying R?SAVE 3-12
Displayword 2-18
Div 2-51
DSF 3-12
DW 2-18

Examples (cont.)
Editfnode 2-32
EF 2-32
H 2-43
Help 2-43
Hex 2-51
LBB 2-44
Listbadblocks 2-44
Miscellaneous commands 2-51
Mod 2-51
Mul 2-51
Restorefnode 2-56
Restorevolumelabel 2-58
Restoring fnodes 3-4, 8
Restoring the volume label 3-11
RF 2-56
RVL 2-58, 3-11
S 2-63
Save 2-60
SB 2-63
Sub 2-51
Substitute word 2-64
Substitutebyte 2-63
SW 2-64
V 2-73
Verify 2-73
W 2-75
Write 2-75
Exit command 2-6, 34

F

File descriptor node (fnode) A-8
File sizes A-19, 21
Fix command 2-6, 35
Fixing bad checksums 2-36
Flexible diskette formats A-21
Flexible diskette track 0 abnormalities A-22
Fnode
Access ID A-12
Altering 2-30
Auxiliary bytes A-13
Backing up on a volume 3-5

Fnode (cont.)
Creation time A-11
data block identification A-11
Displaying 2-23
Flags 2-9, A-9
Freeing 2-38
Granularity A-10
last file access A-11
last modification A-11
Overview A-8
Owner A-11
Parent 2-65, A-13
Restoring 2-54, 3-1, 7
size (bytes) actual data A-11
size (bytes) data space A-12
Structure A-9
Type A-10
Volume blocks A-11
Fnode allocation 2-8
Fnode file 3-1, 2, A-14
Fnode file/space map file inconsistent 2-5
Free command 2-6, 38
Free fnodes map file 2-9, 39, 59, 60, 66, 69, 72, 3-2, A-15
Free space A-15
Free space map file 2-72

G

GB command 2-6, 41
Getbadtrackinfo command 2-6, 41

H

H command 2-7, 43
Help command 2-7, 43
Hex command 2-49

I

Illegal command error 2-4
Initial files A-8
Invocation 1-2
Error messages 1-6
Example 1-5
Interactive 1-6
Single command mode 1-5

IRMX® II volume labels A-4
ISO volume label A-3

L

LBB command 2-7, 44
Listbadblocks command 2-7, 44
Location of files 3-1, A-18, 19
Long files 2-65, 3-1, A-19

M

Manual overview iii
Marking bad blocks 2-8
Miscellaneous commands 2-7, 46
    Add 2-46
    Address 2-46
    Block 2-47
    Dec 2-48
    Div 2-48
    Hex 2-49
    Mod 2-49
    Mul 2-50
    Sub 2-50
Mod command 2- 49
Modes of operation 1-1, 2-1
Mul command 2-50

N

Named volume structure A-1
Named volumes 1-4
Not a named disk error 2-5

O

Operational modes 1-1, 2-1
Orphan fnodes 1-4, 2-36

P

Parameters 2-3
Product overview iii, 1-1

Q

Q command 2-7, 52
Quit command 2-7, 52

R

R command 2-7, 53
R?SAVE 2-11, 14, 27, 33, 54, 55, 57, 3-2, 5, 11, A-17
Radices 2-3
Read command 2-7, 53
Reader Level iii
Reading volume blocks 2-53
Restorefnode command 2-7, 54
Restorevolumelabel command 2-7, 57
Restoring fnodes 3-1
Restoring the volume label 3-10
RF command 2-7, 54
Root directory A-16
RVL command 2-7, 57

S

S command 2-7, 61
Save command 2-7, 59
SB command 2-7, 61
Seek error 2-5
Short files 3-1, A-17
Size of files 2-65, A-19, 21
Space accounting file 3-2, A-15, A-8
Structure of a named volume A-1
Sub command 2-50
Substitutebyte command 2-7, 61
Substituteword command 2-7, 64
SW command 2-64
Syntax error 2-4

T

Track 0 Abnormalities, flexible diskettes A-22

V

V command 2-7, 65
Verify command 2-65
Volume attributes, displaying 1-3, 2-13
Volume blocks, freeing 2-38
Volume free space map file 2-9, 39, 59, 60, 66, 69, 3-2, A-8, 15
Volume label
    backing up 3-6
    iRMX® II A-4
    ISO A-3
    Restoring 3-10
Volume label file 2-57, 3-1, 2, A-8, 16
Volume structure
    Named A-2

W

W command 2-7, 74
Working buffer, changing contents 2-62
Write command 2-7, 74

intel

GUIDE TO THE EXTENDED iRMX®II INTERACTIVE CONFIGURATION UTILITY

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

INTRODUCTION

This manual describes the Interactive Configuration Utility (ICU) and explains its use. It does not explain each screen displayed by the ICU. For a description of the ICU screens and their parameters, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

READER LEVEL

The manual assumes that you are familiar with the monitor and keyboard from which you run the ICU. It is also helpful if you are familiar with the following:

- The Extended iRMX II Operating System
- PL/M-286
- BND286 and BLD286

MANUAL OVERVIEW

This manual is organized as follows:

Chapter 1	This chapter provides introductory material to configuring an iRMX II system using the Interactive Configuration Utility (ICU).
Chapter 2	This chapter describes how to generate a system.
Chapter 3	This chapter describes how to prepare application jobs.
Chapter 4	This chapter provides overview information on how to add users to your system. For detailed information on adding users, refer to the *Extended iRMX Operator's Guide to the Human Interface.*
Chapter 5	This chapter describes how to load and test the system.

Appendix A	This appendix lists files created by the ICU.
Appendix B	This appendix shows an example configuration session.
Appendix C	This appendix describes how to program a generated 286-based system into PROM devices.
Appendix D	This appendix describes how to program a 386/100-based system into PROM devices.

CONVENTIONS

This manual uses the following conventions:

- Information appearing as UPPERCASE characters when shown in keyboard examples must be entered or coded exactly as shown. You may, however, mix lower and uppercase characters when entering the text.
- Fields appearing as lowercase characters within angle brackets (< >) when shown in keyboard examples indicate variable information. You must enter an appropriate value or symbol for variable fields.
- User input appears in one of the following forms:

```
as bolded text within a screen
```

- The term "iRMX II" refers to the Extended iRMX II.3 Operating System.
- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- All numbers unless otherwise stated are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

Intel®

CONTENTS

CHAPTER 1 **PAGE**
INTRODUCTION TO CONFIGURATION

1.1 Introduction 1-1
1.2 What is Configuration 1-1
1.3 When to Use the ICU 1-2
1.4 ICU Location 1-2
1.5 The General Process of Using the ICU 1-3
1.6 What to do Before Invoking the ICU 1-4
1.6.1 Configuration Environments 1-4
1.6.2 Ensuring the ICU Files are on Your Hard Disk 1-5
1.6.3 Choosing Your Definition File 1-7
1.6.4 Checking Access Rights to Definition Files 1-7
1.7 Distinguishing ICU-Generated Files 1-10
1.7.1 Creating Directories for Extended iRMX® II-Based Systems 1-10
1.7.2 Using the Prefix Option 1-11
1.8 Invoking the ICU 1-11
1.8.1 Invocation Error Messages 1-12
1.9 What to do After Invoking the ICU 1-14
1.9.1 Help Command 1-15
1.9.2 Change Command 1-16
1.9.3 Generate Command 1-18
1.9.4 List Command 1-19
1.9.5 Save Command 1-19
1.9.6 Quit Command 1-20
1.9.7 Exit Command 1-21
1.9.8 Replace Command 1-21
1.9.9 Detail-Level Command 1-22
1.9.10 Backup Command 1-23
1.9.11 Aborting ICU Commands 1-24
1.10 Changing a Definition File 1-25
1.10.1 Explanation of the Basic Screen Elements 1-26
1.10.2 Entering File Names, Address Values, and Integer Constants 1-29
1.10.3 Help Messages 1-30
1.10.4 Screen Formats 1-30
1.10.4.1 Fixed Screen Formats 1-30
1.10.4.2 Repetitive Screen Formats 1-30
1.10.4.3 Repetitive-Fixed Screen Formats 1-31

CHAPTER 1 (continued) **PAGE**

1.11 Screen Editing Command for the ICU 1-32
1.11.1 Deleting Data on a Repetitive Screen Format 1-35
1.11.2 Inserting Data on a Repetitive Screen Format 1-37
1.11.3 Deleting a Repetitive-Fixed Screen 1-38
1.11.4 Inserting a Repetitive-Fixed Screen 1-42
1.12 ICU Error Messages 1-42
1.12.1 Interactive Error Messages 1-42
1.12.2 Internal ICU Errors 1-43
1.13 Upgrading Definition Files 1-43
1.14 The ICUMRG Utility 1-45
1.14.1 Invoking ICUMRG 1-47
1.14.2 ICUMRG Example 1-48
1.14.3 ICUMRG Error Messages 1-49

CHAPTER 2 **PAGE**
GENERATING YOUR SYSTEM

2.1 Introduction 2-1
2.2 Generating Configuration Files 2-1
2.3 Executing the Submit File 2-4
2.3.1 Assembling the Configuration Files 2-4
2.3.2 Binding the Individual Subsystems 2-4
2.3.3 Warning Messages 2-5
2.3.4 Building the System 2-5
2.3.5 Error Messages 2-6

CHAPTER 3 **PAGE**
PREPARING APPLICATION JOBS

3.1 Introduction 3-1
3.2 Preparing Application Code 3-1
3.2.1 Language Requirements 3-2
3.2.2 Include Files 3-2
3.3 Determining Memory Locations 3-3
3.4 Binding and Building Your Application Jobs 3-4
3.4.1 BND286 3-4
3.4.2 Minimizing the Memory Address Size 3-6
3.4.3 Building a ROM-Based System 3-7

CHAPTER 4 **PAGE**
ADDING USERS TO YOUR SYSTEM

4.1 Introduction 4-1
4.2 The Resident User 4-1
4.3 Non-Resident Users 4-1

CHAPTER 5 **PAGE**
LOADING AND TESTING THE SYSTEM

5.1 Introduction 5-1
5.2 Loading Your System into RAM 5-1
5.3 Initializing Your System 5-1
5.3.1 Initialization 5-1
5.3.3 Completing Initialization 5-4
5.4 Testing Your System 5-4
5.4.1 Using the Debugging Tools 5-5
5.4.1.1 Advantages of the iRMX® II System Debugger 5-5
5.4.1.2 Advantages of Soft-Scope® 286 5-5
5.4.1.3 Advantages of the I^2ICE™ In-Circuit Emulator 5-6
5.4.2 Debugging Application Jobs 5-6

APPENDIX A **PAGE**
FILES CREATED BY THE ICU

A.1 Introduction A-1
A.2 Created Files A-1

APPENDIX B **PAGE**
EXAMPLE SYSTEM CONFIGURATION

B.1 Introduction B-1
B.2 The Intel-Supplied Definition File B-1
B.3 Differences Between the Target and Start-up Systems B-2
B.4 Steps Performed to Create the Target System B-2
B.5 Using the ICU to Define the Target System B-4

APPENDIX C **PAGE**
PROGRAMMING A 286-BASED SYSTEM INTO PROM DEVICES

C.1 Introduction C-1
C.2 Requirements C-1
C.3 Configuring a ROM-Based System C-2
C.4 Generating/Building the System C-9
C.4.1 Including the iSDM™ Monitor and the Bootstrap Loader in the PROM Devices C-10
C.4.1.1 Setting up the iUP 201 PROM Programmer C-11
C.4.1.2 Formatting the Operating System PROM File C-12
C.4.1.3 Programming the Operating System into PROM Devices C-14
C.4.1.4 Programming the iSDM™ Monitor into PROM Devices C-16
C.4.1.5 Programming the Bootstrap Loader into PROM Devices C-17
C.4.1.6 Starting the Operating System in ROM from the iSDM™ Monitor .. C-17
C.4.2 Creating a System that is activated on Power-up C-18
C.4.2.1 Formatting the Operating System PROM File C-19
C.4.2.2 Programming the Operating System Into the PROM Devices C-20
C.4.2.3 Starting the Operating System in PROM C-23
C.4 Hardware Jumper Modifications C-24

APPENDIX D **PAGE**
PROGRAMMING A 386/100-BASED SYSTEM INTO PROM DEVICES

D.1 Introduction D-1
D.2 Requirements D-1
D.3 Configuring a ROM-Based System D-2
D.4 Generating/Building the System D-9
D.5 Programming the System into PROM Devices D-10
D.5.1 Formatting the Operating System PROM File D-12
D.5.2 Formatting the Copy Routine D-13
D.5.3 Copying the Operating System and the Copy Routine to ROM D-13
D.6 Booting the System D-14
D.7 Considerations for a 386/20-Based System D-15

TABLES

TABLE **PAGE**

1-1 iRMX® II ICU Files 1-6
1-2 Screen Names 1-18
1-3 Integer Constant Formats 1-29
1-4 Special Editing Commands 1-33
2-1 Files Created by the G[enerate] Command 2-3
3-1 Interface Libraries as a Function of PL/M-286 Models 3-5
5-1 Order of Initialization 5-3
A-1 Files Created by the ICU and SUBMIT File A-1

FIGURES

FIGURE		PAGE
1-1	Location of the ICU Directory in an Intel-supplied System	1-2
1-2	First Step: Editing a Definition File	1-3
1-3	Second Step: Generating a Submit File and Source Files	1-4
1-4	ICU Flowchart	1-39
1-5	Merging Intel Device Drivers	1-46
1-6	Merging User Devices	1-47
3-1	Application Job Bind and Build Procedure	3-7
B-1	Memory Maps for the 80286-Based System and Target System	B-3
B-2	Initial ICU Screen	B-5
B-3	The Intel Device Drivers Screen	B-6
B-4	MSC Driver Screen	B-7
B-5	MSC Driver Screen	B-7
B-6	The Intel Device Drivers Screen	B-8
B-7	Query Screen for the iSBC® 220 SMD Device	B-8
B-8	The iSBC® 220 Driver Screen	B-9
B-9	The Completed iSBC® 220 Driver Screen	B-9
B-10	Query Screen for another iSBC® 220 Driver	B-10
B-11	Query Screen for iSBC® 220 SMD Controller Unit Information	B-10
B-12	The iSBC® 220 Unit Information Screen	B-11
B-13	Completed iSBC® 220 Unit Information Screen	B-12
B-14	iSBC® 220 Unit Query Screen	B-12
B-15	iSBC® 220 SMD DUIB Query Screen	B-12
B-16	The iSBC® 220 Device-Unit Information (DUIB) Screen	B-13
B-17	Completed iSBC® 220 Device-Unit Information Screen	B-14
B-18	The Intel Device Drivers Screen	B-15
B-19	iSBC® 208 Device Query Screen	B-15
B-20	The iSBC® 208 Driver Screen	B-16
B-21	Completed iSBC® 208 Driver Screen	B-16
B-22	Query Screen for another iSBC® 208 Driver	B-17
B-23	Query Screen for the iSBC® 208 Unit Information	B-17
B-24	iSBC® 208 Unit Information Screen	B-18
B-25	Completed iSBC® 208 Unit Information Screen	B-19
B-26	iSBC® 208 Unit Query Screen	B-19
B-27	The iSBC® 208 DUIB Query Screen	B-19
B-28	The iSBC® 208 Device-Unit Information (DUIB) Screen	B-20
B-29	Completed iSBC® 208 Device-Unit Information Screen	B-21
B-30	Intel Device Drivers Screen	B-22

FIGURE **PAGE**

B-31 RAM Disk Device Query Screen B-22
B-32 Default RAM Disk Driver Screen B-23
B-33 Inserted RAM Disk Driver Screen B-23
B-34 RAM Driver Query Screen B-23
B-35 RAM Unit Query Screen B-24
B-36 RAM Disk Driver Unit Information Screen B-24
B-37 Inserted RAM Disk Driver Unit Information Screen B-25
B-38 RAM Unit Query Screen B-25
B-39 RAM DUIB Query Screen B-25
B-40 RAM Disk Device-Unit Information Screen B-26
B-41 Inserted RAM Disk DUIB Information Screen B-27
B-42 The Original and Modified Memory Maps B-28
B-43 Memory for System Screen B-29
B-44 Changing the Memory for System Screen B-29
B-45 New Memory for System Screen B-30
B-46 Memory for Free Space Manager Screen B-30
B-47 Changing the Memory for Free Space Manager Screen B-30
B-48 New Memory for Free Space Manager Screen B-31
B-49 Generate File Names Screen B-31
B-50 New Generate File Names Screen B-32
B-51 ICU Menu Screen B-32
B-52 Generation Phase ICU Screen B-33
B-53 Output of Submit File for SAM286.CSD B-35
B-54 Initial ICU Screen B-41
B-55 Memory for System Screen B-42
B-56 Adjusting the Memory for System Screen B-42
B-57 Final Memory for System Screen B-42
B-58 Memory for Free Space Manager Screen B-43
B-59 Changes to the Memory for Free Space Manager Screen B-43
B-60 Final Memory for Free Space Manager Screen B-43
B-61 Entering the List Command B-44
B-62 ICU Menu Screen B-44
B-63 Generation Phase ICU Screen B-45

CHAPTER 1
INTRODUCTION TO CONFIGURATION

1.1 INTRODUCTION

The iRMX II Operating System is modular in structure, enabling you to include or omit subsystems according to your needs. It is also compatible with a variety of peripheral boards. The Interactive Configuration Utility (ICU) is designed to help you take advantage of this flexibility.

This chapter provides an overview of the ICU. It explains the configuration process, configuration environment, ICU files, ICU commands, error messages, the utilities that comprise the ICU, and more. The chapter is very comprehensive and includes many important details for using the ICU. Intel recommends that you read this chapter carefully before attempting to configure your system.

1.2 WHAT IS CONFIGURATION?

Configuration is the process of selecting your application's hardware and operating system layers and then binding and building the entire operating system. The tool used for configuration is the Interactive Configuration Utility (ICU). The ICU is a menu-driven utility which presents a series of screens that prompt you for information. The information is stored in a definition file that is then used to generate the new system.

The objective of running the ICU is to build a definition file that contains all of the configuration information. This file contains two kinds of information:

- Initialization parameters
- A set of variables specifying which operating system layers and device drivers are to be bound together with your application software

Intel provides six definition files you can use as a starting point. If you run the UPDEF Utility supplied with this release, you can also use a definition file from iRMX II.1. (It is not necessary to run UPDEF if you are using a definition file from iRMX II.2.) As you perform the configuration process, you alter the chosen definition file to match your target system.

1.3 WHEN TO USE THE ICU

You should use the ICU whenever one of the following is true:

- You want to generate the configuration files that describe your system.
- You are using a system other than one described by an Intel-supplied definition file.
- You are changing an existing system's hardware and/or software (e.g., adding a new disk drive).

1.4 ICU LOCATION

The ICU files are located in the directory :SD:RMX286/ICU. When working with the ICU, you must use the full pathname in each command (see Figure 1-1) or create an ALIAS for the pathname.

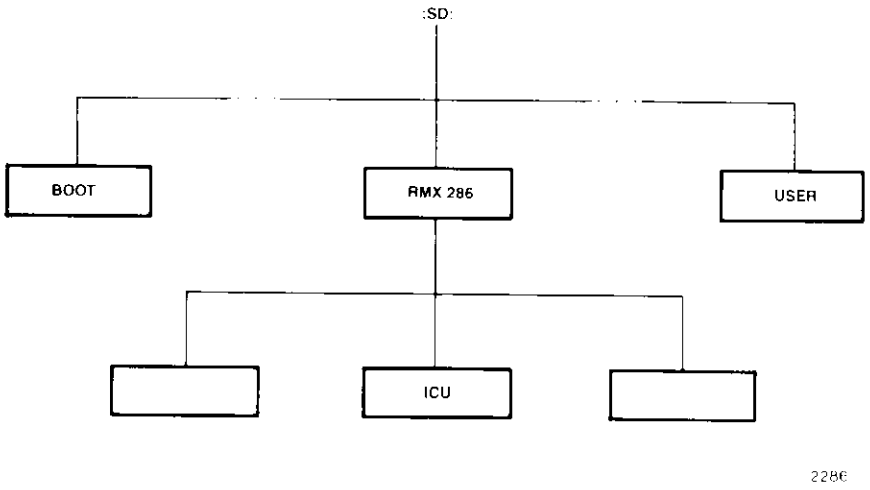


Figure 1-1. Location of the ICU Directory in an Intel-supplied System

1.5 THE GENERAL PROCESS OF USING THE ICU

You configure a system in three steps:

1. Interactively modify a definition file (see Figure 1-2). To do this, invoke the ICU and then supply information to fill in screens that the ICU presents. (This step can be omitted if your system matches one of the Intel-supplied definition files.)
2. When you finish configuring the operating system, use the ICU to generate new configuration files as defined in your modified definition file (see Figure 1-3). The end product is a group of files that define the system.
3. Exit from the ICU, and at the Human Interface level, execute the submit file created by the ICU during the generate step (see Step 2). This creates the new version of the operating system which can then be loaded and executed.

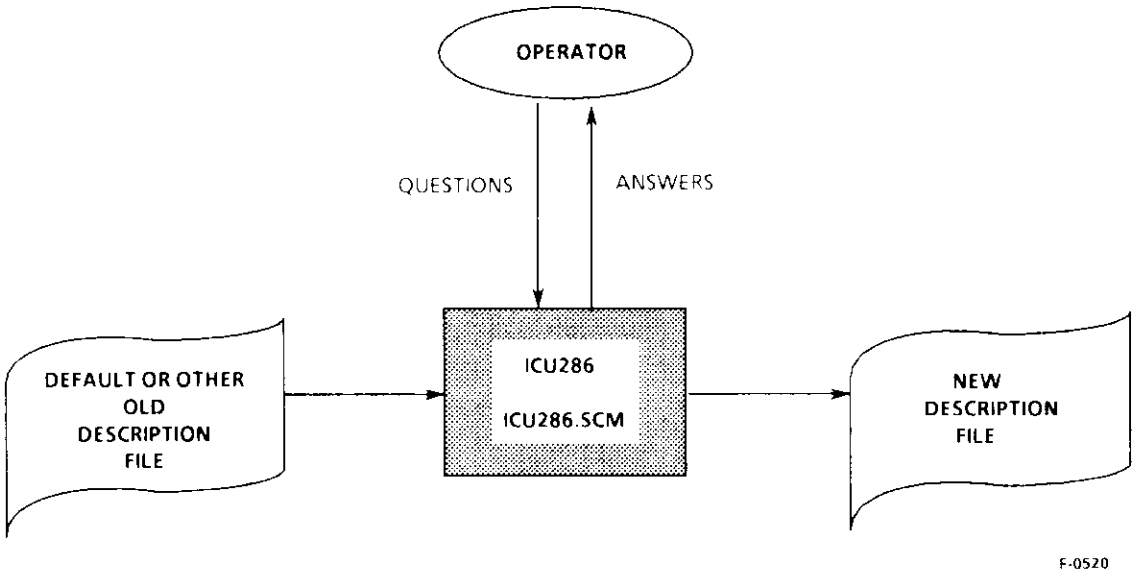


Figure 1-2. First Step: Editing a Definition File

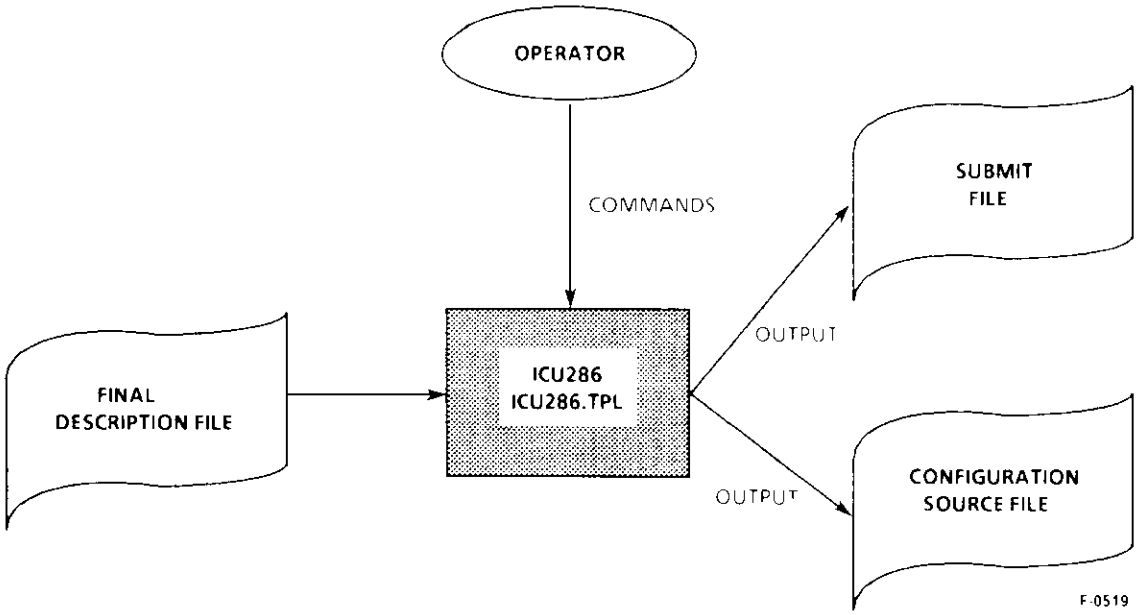


Figure 1-3. Second Step: Generating a Submit File and Source Files

1.6 WHAT TO DO BEFORE INVOKING THE ICU

Before you invoke the ICU, you must perform checks on your existing system and make several decisions. The following sections provide the information you need to know before invoking the ICU.

1.6.1 Configuration Environments

You can run the ICU on the following systems:

- An Intellec® Series IV Microcomputer Development System with 256K contiguous bytes of memory, a hard disk, and version 2.8 or later of the iNDX Operating System.
- An iRMX II-based system with 1 Mbyte of RAM memory which allows a user partition of 384K bytes, a hard disk, and the iRMX II Operating System. For information on changing the amount of memory allocated to your terminal, see the section on editing the :CONFIG:TERMINALS file in Volume I, *Operator's Guide to the Human Interface*.

1.6.2 Ensuring the ICU Files are on Your Hard Disk

Contained on the iRMX II Operating System release diskettes or tape are the files to run the ICU. These files must be on the hard disk before you can invoke the ICU. Follow the instructions in the *Extended iRMX II Hardware and Software Installation Guide* to copy these files to your system.

Table 1-1 lists all of the files required to run the Interactive Configuration Utility for iRMX II systems and for iNDX systems. Check that your hard disk contains all of the files required by your system. If your hard disk does not contain the required files, return to the instructions in the *Extended iRMX II Hardware and Software Installation Guide.* Following the directions in that manual will place all the iRMX II files into the standard directory structure.

Table 1-1. iRMX® II ICU Files

Function	Filename
ICU286	ICU286 (for iRMX II systems) ICU286.86 (for iNDX systems)
Screen Master File	ICU286.SCM
Template File for System Generation	ICU286.TPL
Update Definition Utility	UPDEF (for iRMX II systems) UPDEF.86 (for iNDX systems)
User Device Support Utility	UDS (for iRMX II systems) UDS.86 (for iNDX systems)
UDS Screen Master File	UDS.SCM
Template example - (minimum UDS input file)	TEMPLATE_1.UDS
Template example - (UDS input file containing user help built into help text)	TEMPLATE_2.UDS
ICUMRG Utility	ICUMRG (for iRMX II systems) ICUMRG.86 (for iNDX systems)
Definition File for an iRMX II Multi-User System designed to run on the iSBC 286/10(A) and iSBC 286/12 boards	28612.DEF
Definition File for an iRMX II Multi-User System designed to run on the SXM 386 AP Kit	SXM 386.DEF
Definition File for an iRMX II Multi-User System designed to run on an iSBC 386/2X and 386/3X board	38620.DEF
Definition File for an iRMX II Multi-User System designed to run on an iSBC 286/100A board	286100A.DEF
Definition File for an iRMX II Multi-User System designed to run on an iSBC 386/116 and 386/120 board	386100.DEF

1.6.3 Choosing Your Definition File

If you have never configured an iRMX-based system before, you should choose one of the Intel-supplied, multi-user definition files (listed in Table 1-1) as input into the ICU. You can build a definition file screen by screen, but you will save time by starting with a standard definition file. Details on the standard definition files are given in the *Extended iRMX II Hardware and Software Installation Guide*.

If you created a definition file using an iRMX II.1 version of the ICU, you can use this file as input to iRMX II.3 of the ICU only after running the UPDEF Utility. Once you create a iRMX II.3 definition file, you cannot use it as input into iRMX II.1 of the ICU.

The Intel-supplied definition files define 80286- or 80386-based MULTIBUS I and MULTIBUS II systems. These systems are fully configured multi-user iRMX II Operating Systems. Each layer implements all its system calls and most of the features and drivers provided by the iRMX II Operating System. Multiple users can communicate with the operating system interactively through a terminal or via an application program, and can access secondary storage. The definition files include UDI so that you can run languages, such as PL/M-286, PASCAL, and FORTRAN.

To define your own system, modify the definition file that comes closest to your needs. To see the contents of a definition file, use the LIST command (described later in this chapter). Details of board configuration and interrupt levels are given in the *Extended iRMX II Hardware and Software Installation Guide*.

1.6.4 Checking Access Rights to Definition Files

If you use the ICU on an iRMX II-based system, the operating system makes sure you have the proper access to both the definition files and their respective directories. This check is performed in two instances: when you invoke the ICU and when you enter the G (generate) command (discussed in a later section). If you do not have proper access, one or more of the following situations can occur:

- The ICU will be unable to read the input definition file.
- The ICU will be unable to save the changes you make during the ICU session.
- The ICU will be unable to create the generation files necessary to complete the configuration process.

Following the installation instructions in the *Extended iRMX II Hardware and Software Installation Guide* ensures that the user WORLD can generate a new version of the operating system.

To check your access rights to directories and files contained in directories, use the DIR command followed by the E[xtended] or L[ong] options. For example, you can check your access rights to the :SD:RMX286/ICU directory and the 28612.DEF file by using the following commands:

```
DIR :SD:RMX286 E[xtended]
DIR :SD:RMX286/ICU E[xtended]
```

For more information on using the Human Interface commands, see the *Operator's Guide to the Extended iRMX II Human Interface*.

The access rights needed to use the ICU successfully vary according to the operations to be performed. In all cases where the G (generate) command is to be used, you must have Add Entry access to the directory containing the definition file and to the directories you specify in the "Generate File Names" screen. In other cases, the access required depends on the kind of file (new or existing) and whether it is an input or output file. The following paragraphs describe the access rights required in different circumstances.

- If you specify an existing definition file as an input file, you must have Read access to the definition file.
- If you specify an existing definition file as an output file, you must have Delete and Write access to the definition file. You must also have Add Entry access to the directory containing the definition file.
- If you specify an existing definition file as the only definition file on the command line, the file serves as both an input and an output file. In this case, you must have Read/Write and Delete access to the definition file. (Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for more information about the "Generate File Names" screen.)
- If you specify a new file as the only file on the command line, the file serves as both an input file and an output file. In this case, you must have Add Entry access to the directory containing the new file.

If you do not have the correct access rights, the ICU returns the following message:

```
0026: E$FILE_ACCESS, while loading command
```

Additional information about access rights can be found in the *iRMX II Extended I/O System User's Guide*.

EXAMPLES:

To use the G command without causing an error, you must have Add/Delete Entry access to the directory in which you are working. This section describes the various ways of invoking the ICU and the access rights required.

- /RMX286/ICU/ICU286 :HOME:NEW.DEF

where

:HOME:NEW.DEF — Pathname of a new definition file. The ICU uses this file as both the input file and the output file.

You must have Add Entry access to the directory containing NEW.DEF (:HOME:). If you do not have Add Entry access, the ICU returns the following message:

```
*** I/O Error in file: :HOME:NEW.DEF
0026: E$FILE_ACCESS
```

If the directory does not exist, the ICU returns the following message:

```
*** I/O Error in file: :HOME:NEW.DEF
0021: E$FILE_NOT_EXIST
```

- /RMX286/ICU/ICU286 DIR/OLD.DEF

where

DIR/OLD.DEF — Pathname of an existing definition file. The ICU uses this file as both an input and an output file. It can be a new or existing file.

You must have Read/Write and Delete access to save changes to OLD.DEF.

- /RMX286/ICU/ICU286 /RMX286/ICU/INPUT.DEF to DIR/OUTPUT.DEF

where

/RMX286/ICU/INPUT.DEF	Pathname of a standard definition file. The ICU uses this file as the input file.
DIR/OUTPUT.DEF	Pathname of the output definition file. This file can be a new or existing file. You must have Add Entry access to the directory containing OUTPUT.DEF (DIR). In addition, if OUTPUT.DEF is an existing file, you must have Delete access to it.

1.7 DISTINGUISHING ICU-GENERATED FILES

Each time you generate your system, the ICU generates a set of ICU files. To help you distinguish your generation files from each other and to determine which input definition file generated the ICU files, you can use one of these options:

- Create a new directory to contain your definition file.
- Use the prefix option supplied by the ICU.

The following sections describe each method in more detail.

1.7.1 Creating Directories for iRMX® II-Based Systems

Intel recommends that you maintain the default directory structure by placing any new system definition files in a new directory nested in your :HOME: directory. Before invoking the ICU, you should create a copy of the input definition file in your working directory to avoid corrupting the original file. The following example illustrates how to do this using the 28612.DEF file as the starting definition file:

```
-
-
```

Once you attach the new directory (28612) as the working directory, you will need to use the full pathname to invoke the ICU. For example

You may wish to create an ALIAS for the complete pathname used to invoke the ICU. A convenient convention to use is to create the working directory with the same name as the definition file (without the .DEF extension). The operating system produced should have the same name as the definition file (with a .286 extension).

1.7.2 Using the Prefix Option

A second method to distinguish your ICU generated files from each other is to use the prefix option supplied by the ICU. You can select the prefix option when entering the Generate (G) command. The ICU then displays a prompt (see Chapter 2 for the actual screen) asking you for the prefix letter you wish to assign to the files created by the ICU. For example, if you choose the letter "Q" as your prefix option, a "Q" will precede all the files generated when you enter the Generate command on the menu screen. In this case, the files generated for the Nucleus will be

QNTABL.A28
QNUCDA.A28
QNJOBC.A28

The files created for all other subsystems will also be preceded by the letter "Q" as in the example above. If you want to generate configuration files for more than one system, choose a different prefix option each time. Intel also recommends that your input definition file start with the same prefix letter you assign to the generated files. This allows you to easily determine which definition files created each set of output files. If you create a file with a prefix that already exists, the ICU overwrites the previous file.

If you do not want to use the prefix option, enter a carriage return when you are prompted for the prefix. This causes the ICU to generate the output files without a prefix (see Appendix A for a complete list of the ICU generated files).

1.8 INVOKING THE ICU

This section lists the syntax necessary to invoke the ICU. When invoking the ICU, be sure that the ICU286.SCM file and the ICU286.TPL file reside in the same directory as the ICU286 file that you are invoking.

The syntax for invoking the ICU is as follows (brackets indicate optional items):

ICU286 [input-file-name TO] output-file-name

where

input-file-name	Name of the definition file from which the ICU obtains configuration information. This is typically an Intel-supplied definition file or a definition file from a previous use of the ICU.
output-file-name	Name of the definition file to which the ICU writes updated configuration information.

The following guidelines must be followed when specifying input and output files:

input-file-name: If both an input-file-name and an output-file-name are specified, the input-file-name must represent an existing definition file created by the ICU.

output-file-name only: If the input-file-name is omitted, you enter only the output-file-name, the ICU uses the output-file-name as both the input and output definition file. When the ICU session is complete, the ICU writes the updated configuration information back to the output-file-name.

The output-file-name entered can represent a new or an existing definition file. If the output-file-name specified does not exist, the ICU searches the directory containing ICU for a file named ICU286.DEF. If ICU286.DEF exists, the ICU uses it as the input file. In any case where an input file does not exist, the ICU displays the following message among the main screen display:

```
NEW CONFIGURATION FILE
```

and the session starts with the ICU default values. After saving or exiting, the edited file is stored in the named output file.

1.8.1 Invocation Error Messages

When issuing the invocation previously described, a number of error messages can occur. These error messages are described in the following paragraphs.

Invoking the ICU with no parameters or invalid parameters causes the following message to be displayed:

```
*** INVALID INVOCATION ***
USAGE:  ICU286  [input-file TO] output-file
```

Invoking the ICU with a corrupted definition file, or a file that is not a definition file, causes the following message to be displayed:

```
*** ERROR - FILE: <file_name> IS NOT VALID
```

On invocation the ICU validates the file version numbers. ICU286.SCM, ICU286.TPL, and each definition file have an Intel Version Number, an Update Version Number, and a User Version Number. The Intel version number changes whenever Intel upgrades the ICU to support new release. The Update version number changes whenever the ICU is upgraded, using the ICUMRG utility, to support an update; and the User version number changes whenever a user device is added (using the UDS and ICUMRG utilities).

The ICU checks the version numbers when it is invoked. If there is an inconsistency in the version numbers of either ICU286.SCM or ICU286.TPL, the ICU displays the following error message and returns control to the operating system:

```
*** ERROR - INCONSISTENCY IN THE VERSION OF THE INTERNAL ICU FILES
    Versions:           Intel   Update      User
            ICU286.SCM   <Intel> + <Update> <User version>
            ICU286.TPL   <Intel + Update> <User version>
```

If the inconsistency is in the definition file, the ICU displays the following warning message and asks for permission to upgrade the file or restore from the file. (The upgrade process is a simpler, faster operation than restore and requires no additional user input.)

```
*** WARNING - DEFINITION FILE VERSION IS NOT CORRECT.

                            Intel   Update  User
            ITS VERSION IS:    <the inconsistency>
            VERSION EXPECTED:  <correct version>
```

If the ICU needs to restore from a file in order to use it, you will be prompted as follows:

```
Do you want to restore from the file? y/[n]
```

A response of "No" causes the ICU to stop executing. A "Yes" response means the ICU will restore the backup information stored in the definition file (discussed later in this chapter). If the Update version number is higher than the ICU version number, you are probably using the wrong version of the ICU. In this case, the ICU displays this warning before the restore prompt:

```
*** WARNING - The Definition File version is NEWER
```

However, if the ICU is able to use the file without restoring, it prompts with

```
Do you want to update the file? y/[n]
```

A response of "No" causes the ICU to stop executing. A "Yes" response causes the ICU to update the file. Since the ICU processes definition files with inconsistent version numbers, you can use all of the Intel-supplied definition files as input for your own tailor-made ICU.

The ICU issues the restore prompt if it discovers any of the following in the definition file:

- Inconsistency in the Intel version number.
- Inconsistency in the User version number, if the file contains user devices added with UDS and ICUMRG.
- Inconsistency in the Update version number, if the file version number is higher (newer) than the ICU version number.

The ICU prompts for permission to use the file without restoring in the following cases:

- Inconsistency in the User version number, if the file does not contain user devices.
- Inconsistency in the Update version number, if the file version number is smaller than the ICU version number.

1.9 WHAT TO DO AFTER INVOKING THE ICU

After you invoke the ICU, the main menu screen is displayed as follows:

```
iRMX II Interactive Configuration Utility For Extended iRMX II,<v>
Copyright <years> Intel Corporation

 For general help in any screen enter H <CR>.

The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND :
```

In the screen shown above is the main ICU menu, whenever you see this screen you are in command mode. The string <v> represents the ICU version number. The string <years> represents the copyrighted years of the product.

Whenever you are in command mode, you must enter one of the commands listed or an "H" for help. All of your responses should be followed by a carriage return. The ICU regards all invalid input as a response of "H <CR>" and displays the "Help" screen until a valid response is entered.

The following sections describe the choices on the menu screen.

1.9.1 Help Command

```
ENTER COMMAND : H <CR>
```

If you enter H (help) and a carriage return, the ICU will display the following screen:

```
The Change (C) command allows you to specify the configuration
parameters that define your system.  To get to a specific
screen, type 'C screen$name'.  C? gives you a list of all the
screen names.

The Generate (G) command creates the submit file and all PLM
and assembler files required to create your Extended iRMX II
system.

The List (L) command shows you the current state of your
configuration file.  To copy the values that define your
system to a file, type 'L file$name'.

The Quit (Q) command leaves the ICU without saving any
changes.

The Exit (E) command leaves the ICU and saves all the
changes.

The Save (S) command saves all the changes without leaving
the ICU.

The Replace (R) command replaces the current control
character.

The Detail level (D) command sets the level of detail.

The Backup (B) command writes a backup file.

TYPE <CR> to Continue
```

1.9.2 Change Command

```
ENTER COMMAND : C[hange] [Screen Abbrev] <CR>
```

The Change command enables you to begin editing the definition file. The syntax of the Change command is as follows (the elements inside the brackets are optional):

C[hange] [screen abbrev]

or

C ?

where

C or Change	Starts editing the definition file from the first screen (the "Hardware" screen). The first time you run the ICU you should use this option.
screen abbrev	Begins editing at a specific screen. For example, if you enter "C", a space, the abbreviation of an existing screen, and a carriage return, the ICU enables you to start editing your definition file from that particular screen. If you enter a screen abbreviation incorrectly, the ICU displays a screen containing all the screen names and abbreviations (see Table 1-2). The abbreviation enclosed in parentheses indicates what must be entered for each screen.
?	Causes the ICU to display a screen with all the screen names and abbreviations.

Table 1-2 lists all the possible screen names. The screens are displayed in order from left to right, that is the "Interrupts" screen is displayed after the "Hardware" screen. Device drivers are listed at the end of the table.

If you did not invoke the ICU with the name of an existing definition file, you should start your edit with the "Hardware" screen. If you did invoke the ICU with the name of an existing definition file, you can start your edit with the name of any screen that the input definition file has already defined. If you enter a valid screen name but that screen is not configured into your definition file, the ICU displays the next "main" screen followed by this warning:

```
***Warning - The screen requested cannot be displayed
```

The ICU progresses from screen to screen in a logical order. Refer to Figure 1-4 for the logical flow of the ICU.

Table 1-2. Screen Names

----- Main screens: -----	(HARD) Hardware	(MBII) Multibus II Hardware
(INT) Interrupts	(SLAVE) Slave Interrupt	(MEMS) Memory for System
(MEMF) Memory for FSM	(SUB) Sub-systems	(OSEXT) OS Extensions
(HI) Human Interface	(HIJOB) HI Jobs	(RES) Resident/Recovery User
(PREF) Prefixes	(HILOG) HI Logical	(APPL) Application Loader
(REM) Remote file Access	(REMFS) Remote Servers	(EIOS) EIOS
(ABDR) Auto Boot Dev	(LOGN) Logical Names	(IOUS) I/O Users
(IOJOB) I/O Jobs	(BIOS) BIOS	(BCALL) BIOS System Calls
(IDEVS) Intel Devices	(USERD) User Devices	(UDDM) UDS Device Driver Mods
(SDB) System Debugger	(NUC) Nucleus	(NCOM) Communication Service
(USERJ) User Jobs	(USERM) User Modules	(ROM) ROM Code
(INCL) Includes and Libs	(GEN) Generate File Names	(COMNT) Comments screen

----- Device Drivers -----		(D214) Mass Storage Controller Driver
(D8274) 8274 Terminal Driver		(D8251) 8251A Terminal Driver
(D2530) 82530 Terminal Driver		(D534) 534 Terminal Driver
(D544) 544A Terminal Driver		(D8848) Terminal Comm controller
(D286) Line Printer - iSBC 286/10		(D350) Line Printer - iSBX 350
(D220) iSBC 220	(D218) iSBX 218A	(D202) iSBX 208
(D264) iSBC 264	(D251) iSBX 251	(DRAM) RAM Disk Driver
(DSCSI) SCSI Driver	(D224A) iSBC 186/224A	(D410) iSBC 186/410
(D279) iSBX 279		

ENTER screen abbreviation:

1.9.3 Generate Command

```
ENTER COMMAND : G[enerate] <CR>
```

The Generate command creates all the ASM, PL/M, build and submit files required to configure the iRMX II system. Refer to Chapter 2 for more information on generating a system.

1.9.4 List Command

```
ENTER COMMAND : L[ist] [name] <CR>
```

The List command enables you to list the contents of your definition file to a file or to a device. This command lists the contents of those screens that you selected to define your system. The syntax of the List command is as follows (the elements inside the brackets are optional):

L[ist] [name]

where

L or List	Lists the contents of your screens.
name	Specifies an iNDX or iRMX II device or file. If you omit the name, the terminal (:CO:) is assumed. You should list the definition file to a file name rather than to the terminal since the display scrolls rapidly. If you want to use your terminal to review your definition file, use the Change command to view just those screens you want.

After the ICU has listed the definition file to the specified filename, it notifies you that the definition file has been listed and returns to command mode. For example, if you listed the ICU screens to a file called ICU286.LST, the ICU would display

```
The Definition File has been listed to file: ICU286.LST
```

followed by the main menu screen.

1.9.5 Save Command

```
ENTER COMMAND : S[ave] [name] <CR>
```

The Save command updates your definition file with all of the changes you entered during the current ICU session. The syntax of the Save command is as follows (the elements inside the brackets are optional):

S[ave] [pathname]

where

S or Save	Saves all the changes made in this session.
pathname	Pathname of a file to use instead of the default output-file-name to save changes to the definition file.

When the Save command is entered (followed by a carriage return), the ICU updates the file you specified as the output-file-name. After the ICU updates the definition file, it notifies you that the specified file has been updated and returns to command mode. For example, if you invoked the ICU using 28612.DEF as the output-file-name, the ICU would display this message followed by the menu screen:

```
The Definition File has been written to file:  28612.DEF
```

To be sure you are updating the right file, use the List command before you save your definition file. The List command displays the name of the output definition file at the top of each ICU screen.

1.9.6 Quit Command

```
ENTER COMMAND : Q[uit] <CR>
```

The Quit command enables you to stop your current ICU session without updating the definition file. The syntax of the Quit command is as follows (the elements inside the brackets are optional):

Q[uit]

After you enter the Quit command (followed by a carriage return), the ICU may display the prompt "Do you want to quit without saving your changes? y/[n]" to ensure that you did not accidentally enter the Quit command. Your response to this prompt should be either "Yes" or "No". The ICU only displays this prompt if you use the Quit command after making changes to an existing definition file or creating a new definition file. If no changes were made to the definition file before the Quit command was entered, no prompt is displayed.

1.9.7 Exit Command

```
ENTER COMMAND : E[xit] [pathname] <CR>
```

The Exit command exits the ICU and updates the definition file with all of the changes from the current ICU session. The syntax of the Exit command is as follows (the elements inside the brackets are optional):

E[xit] [pathname]

where

E or Exit	Exits the ICU saving all the changes made in this session.
pathname	Pathname of a file to use instead of the output-file-name to save changes made to the definition file.

You should always use either the Exit or Save command after using the Generate command.

1.9.8 Replace Command

```
ENTER COMMAND : R[eplace] <CR>
```

The Replace command enables you to change the control character that the ICU uses in the special editing commands. The control character precedes special editing commands, the default character is the caret (^). If your terminal does not support this character, or you prefer a different character, use R[eplace] to change it to any character of your choice. The syntax of the Replace command is as follows (the elements inside the brackets are optional):

R[eplace]

After you enter the Replace command (followed by a carriage return), the ICU displays the following screen:

```
ENTER COMMAND : R[eplace] <CR>

Input new control character :
```

Enter the new control character you select, followed by a carriage return.

1.9.9 Detail-Level Command

```
ENTER COMMAND : D[etail-Level] <CR>
```

The Detail-Level command enables you to set the level of detail you want in displaying the ICU screens. This command provides the option of selective screen displays. Rather than viewing all the screens, you can elect to see only screens of a particular type. There are four possible levels you may request:

All	Shows all the screens
Devices	Shows only device screens
Operating System	Shows all non-hardware related screens
Jobs	Shows only the job screens (such as User, I/O and User Modules screen)

The syntax of the Detail-Level command is as follows (the elements inside the brackets are optional):

D[etail-Level]

After you enter the Detail-Level command (followed by a carriage return), the ICU displays the following screen:

```
The following levels of detail are available:

All
Devices
Operating-System
Jobs

ENTER Level of Detail :
```

If you enter an invalid response, the ICU redisplays this screen until it receives a valid response.

1.9.10 Backup Command

```
ENTER COMMAND :  B[ackup] file-name <CR>
```

The Backup command writes an ASCII backup file containing a list of all the parameter abbreviations and their current values. The backup file is used as input to the ICU during the restore process (discussed later in this chapter). Remember, the information in the backup file is part of the definition file. The advantage of creating a backup file is that it is in ASCII which is easier and safer to use with other utilities or electronic mail.

The syntax of the Backup command is as follows (the elements inside the brackets are optional):

B[ackup] filename

where

B or Backup	Writes a backup file.
filename	Name of the file that will contain the backup information.

When the backup command has completed, the ICU displays a message that the filename specified has been backed-up. It then returns to command mode. For example, if you backed-up your definition file to a file named UPDATE.BCK, the following screen would be displayed.

```
 The Definition File has been backed-up to file: UPDATE.BCK


 For general help in any screen enter H <CR>.

The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND :
```

1.9.11 Aborting ICU Commands

The ICU enables you to abort an ICU process, without losing any information, by entering CONTROL-C. If you enter CONTROL-C during the execution of an ICU command (Generate, List, etc.) the ICU stops executing the current command, and returns to the main menu screen. The ICU handles CONTROL-C differently for each command.

- If entered in command mode, or during SAVE, QUIT, EXIT or BACKUP, it is ignored.
- If entered during CHANGE, it displays the following message:

```
'Type C to EXIT to the Main-Menu'
```

- If entered during GENERATE, the ICU finishes writing the file being generated, displays

```
'*** Process ABORTED.'
```

and returns to command mode.

- If entered during LIST, the ICU displays the Process ABORTED message, writes the message to the file or device specified in the List command, and returns to command mode.
- If entered during REPLACE or DETAIL-LEVEL, the ICU returns to command mode.
- If entered while restoring, the ICU displays the following message and returns control to the operating system.

```
'***  Process ABORTED - The Definition File was not restored.'
```

1.10 CHANGING A DEFINITION FILE

It is possible to change a definition file by entering the "Change" command on the menu screen. The ICU then shows one screen of information at a time. Each screen pertains to a specific area of configuration. The information displayed on the screen consists of a series of prompts and default values. Any of the default values can be changed. However, the changes you make are not immediately displayed on the screen. They are displayed only when you reshow the screen using the editing command ^R (or R), discussed later in this chapter, or just a carriage return.

Entering another carriage return after using one to reshow a screen causes you to proceed to the next screen. The changes are recorded in the definition file when you exit the ICU using the "E" command or when you enter the "S" command while still in the ICU.

1.10.1 Explanation of the Basic Screen Elements

The following definitions will help you understand the various parts of a screen. The following screen illustrates the defined terms.

```
(SCABV)   SCREEN NAME

(ABV) PARAMETER DEFINITION [range of values]      XXX
(ABV) PARAMETER DEFINITION [range of values]      XXX

                          •
                          •
                          •


Enter [Abbreviation = new value / Abbreviation ? / H ] :
<prompt line>
```

(SCABV)	The abbreviation enclosed in parentheses identifies the screen being displayed. This abbreviation is used with the "Change" or "Find" (discussed later in this chapter) commands to access a screen.
SCREEN NAME	The name of the screen.
(ABV)	The abbreviation enclosed in parentheses identifies the parameter whose existing value can be replaced.
PARAMETER DEFINITION	This definition briefly describes the parameter that you can change.
[range of values]	This defines the range of acceptable values for this parameter.
XXX	The value in the current definition file. If the existing value is not what you want, replace it with any other value within the range of values.

<prompt line>

This line is where you enter changes to the screen. The cursor is located at the beginning of this line ready for you to enter one of the following:

- An abbreviation, an equal sign (=) and a new value
- An abbreviation and a "?", if you need an explanation of the parameter
- A ^H (H), if you need general help in understanding the screen types or editing commands
- A "?", if you need an explanation of the specific screen

Data you enter on the prompt line should be followed by a carriage return (<CR>).

The following screen is the first screen displayed when entering Change mode with a new definition file. All of the features described above are displayed. The screen abbreviation is (HARD) and the screen name is "Hardware". There are nine (9) parameter lines and a prompt line. Each parameter line includes a range of legal values which may be entered if the default value does not meet your system requirements. The bolded entries on the following screen illustrate how you would use the prompt line to make changes to two parameter lines.

The hardware screen chapter of the *Extended iRMX II Interactive Configuration Utility Reference Manual* explains how to respond to the specific prompts shown in this screen. The purpose of this section is to explain how to make entries on this and other types of screens.

```
(HARD)    Hardware

(BUS) System Bus Type [1 = MBI / 2 = MBII]          1
(TP)  8254 Timer Port [0-0FFFFH]                    0D0H
(CIL) Clock Interrupt Level [0-7]                   0
(CN)  Timer Counter Number [0,1,2]                  0
(CIN)  Clock Interval [0-65535 msec]                10
(CF)  Clock Frequency [0-65535 khz]                 1229
(TPS) Timer Port Separation [0-0FFH]                02H
(NPX) Numeric Processor Extension [Yes/No]          YES
(IF)  Initialize On-board Functions [1,2,3,4/No]    1
(BIP) Board Initialization Procedure [1-45 Chars]

Enter [Abbreviation = new value / Abbreviation ? / H ] :
:cil=4 <CR>
:npx=no<CR>
<CR>
```

```
(HARD)    Hardware

(BUS) System Bus Type [1 = MBI / 2 = MBII]          1
(TP)  8254 Timer Port [0-0FFFFH]                    0D0H
(CIL) Clock Interrupt Level [0-7]                   4
(CN)  Timer Counter Number [0,1,2]                  0
(CIN)  Clock Interval [0-65535 msec]                10
(CF)  Clock Frequency [0-65535 khz]                 1229
(TPS) Timer Port Separation [0-0FFH]                02H
(NPX) Numeric Processor Extension [Yes/No]          NO
(IF)  Initialize On-board Functions [1,2,3,4/No]    1
(BIP) Board Initialization Procedure [1-45 Chars]

Enter [ Abbreviation = new value / Abbreviation ? / H ]
:
```

1.10.2 Entering File Names, Address Values, and Integer Constants

You can enter several types of values in response to a parameter line, depending on the range of values for the parameter. The kinds of values you can enter include

device/file name — A device or file name can be any device or file name acceptable to the operating system.

integer constants — Constants must be unsigned integers that you can enter in any of three radices: decimal, hexadecimal or kilobyte. A trailing radix character indicates the radix of the number, as shown in Table 1-3. The default radix is decimal.

addresses — Address values must be entered in the form SELECTOR:OFFSET. The radix must be specified (either explicitly or by default) for both portions of an address. For example, you must specify the selector of 900H and an offset address of 384H as 900H:384H.

Table 1-3. Integer Constant Formats

Radix	Trailing Character
Decimal	None or D
Hexadecimal	H or h
Kilobytes	K or k

1.10.3 Help Messages

The ICU provides three types of help messages to supply information and save you time as you are defining your definition files.

- For HELP about parameters, enter the parameter abbreviation followed by a "?".
- For HELP about the screen being displayed, enter a ?.
- For HELP about editing screens, enter ^H or H.

After reading the help messages, enter a carriage return to return to the screen you were editing.

1.10.4 Screen Formats

Three basic types of screen formats are used in the ICU: the fixed screen, the repetitive screen, and the repetitive-fixed screen. These screen formats have similar features.

1.10.4.1 Fixed Screen Formats

The fixed screen format enables you to make changes by entering the two- or three-letter abbreviation, the equal sign (=), the new value, and a carriage return. The "Hardware" screen shown earlier in this chapter is a fixed format screen.

1.10.4.2 Repetitive Screen Formats

Most screens use the fixed format to display information. However, a screen such as the "Prefixes" screen, shown below, uses a repetitive screen format. In a repetitive screen format, the same prompt is repeated many times. Each time you enter information on the screen, you define new system information. In the example below, each time you enter a line of information, you define a logical name for a directory. As you can see from the example, identifying numbers precede each line of information. To make changes to this screen, you should enter the line number, the equal sign (=), the new value, and a carriage return. After entering the change, the screen is redisplayed.

```
(PREF)     Prefixes

    Prefix = 1-45 characters
[1] Prefix = :PROG:
[2] Prefix = :UTILS:
[3] Prefix = :SYSTEM:
[4] Prefix = :LANG:
[5] Prefix = :$:
[6] Prefix =

Enter Changes [Number = new value / ^D Number / ? / H ]:
```

1.10.4.3 Repetitive-Fixed Screen Formats

The repetitive-fixed screen format combines the features of the other two screen types. It repeats a full screen of information any number of times. In the following example, the "User Jobs" screen, you define a user job by entering information on the screen. When you complete this screen or any repetitive-fixed screen, a one-line query screen is displayed. In this case the query screen asks: "Do you have any/more User Jobs?". If you answer "yes" or "y", the ICU presents another "User Jobs" screen. Each time you make entries to one of these screens you define a new user job. The ICU repeats this screen until you respond with a "no", "n", and/or a carriage return to the prompt.

```
(USERJ)     User Jobs

(NAM) Job Name [0-14 Char]
(ODS) Object Directory Size [0-3840]                    0
(PMI) Pool Minimum [20H-0FFFFFH]                        060H
(PMA) Pool Maximum [20H-0FFFFFH]                        0FFFFFH
(MOB) Maximum Objects [1-0FFFFH]                        0FFFFH
(MTK) Maximum Tasks [1-0FFFFH]                          0FFFFH
(MPR) Maximum Priority [0-255]                          129
(EHS) Exception Handler Entry Point [1-31 Chars]

(EM)  Exception Mode [Never/Prog/Environ/All]           NEVER
(PV)  Parameter Validation [Yes/No]                     YES
(TP)  Task Priority [0-255]                             155
(TSA) Task Entry Point [1-31 Chars]

(VAR) Public Variable Name [0-31 Chars]

(SSA) Stack Segment Address [SS:SP]                     0000:0000H
(SSI) Stack Size [0-0FFFFH]                             0300H
(NPX) Numeric Processor Extension Used [Yes/No]         NO

Enter [Abbreviation= new value / Abbreviation ? / H ]
: <CR>
```

1.11 SCREEN EDITING COMMANDS FOR THE ICU

Several special commands are available to simplify the editing process. They are summarized in Table 1-4 and then explained in detail in the following paragraphs. The commands are initiated by entering the caret "^" control character (or a character you substituted for the caret using the "R" command in command mode) followed by one or more characters. It is also possible to enter all of the commands, except Insert, Copy, and Delete, without the control character. If you try to use Insert or Delete without the control character, you will receive a message explaining the correct invocation of these commands. Each command sequence must be terminated with a carriage return.

Table 1-4. Special Editing Commands

Command	Meaning	Screens Affected		
		Fixed	Repet-itive	Repet-fixed
^B or B	Back up to previous screen	X	X	X
^C or C	Return to command mode	X	X	X
^D	Delete a screen			X
^D <number>	Delete the element with this number		X	
^F <scabv> or F <scabv>	Find and display the specified screen	X	X	X
^H or H	Display the list of special commands that apply to the current screen format	X	X	X
^I	Insert an new screen in front of the current screen			X
^I <number>	Insert a new line		X	
^CO	Copy the current screen			X
^R or R	Redisplay the current screen	X	X	X
^N or N	Go to the next logical screen	X	X	X
^S <string> or S <string>	Search the remaining screens for the specified string			X

Complete descriptions of the special editing commands are as follows:

^B or B — Enables you to move backwards from the current screen to the previous screen. The ICU displays the previous screen and enables you to continue as usual. Moving backwards beyond the beginning of the definition file returns you to command mode. This command can be used on all types of screens.

^C or C — Returns you to command mode from any ICU screen. It then displays the main menu.

^D — Enables you to delete an entire repetitive-fixed format screen. The screen deleted is the current screen.

^D <number>	Enables you to delete a specific item in a repetitive screen. The number you enter identifies the entry to be deleted.
^F <scabv> or F <scabv>	Finds and displays the screen indicated by the screen abbreviation. The syntax of the ^F command is ^F (or F) screen-abbreviation where the screen-abbreviation can be any abbreviation listed in Table 1-2. This command enables you to jump from one screen to another. If you specify a screen name not previously defined, this command jumps to the next available screen, and displays this warning message: `*** WARNING - The screen requested cannot be displayed` If you do not specify a screen abbreviation, the list of screen names and abbreviations is displayed (see Table 1-2) and you are prompted for a screen abbreviation. If you want to exit this command without entering an abbreviation, press the carriage return and continue to the next logical screen. Figure 1-4 shows a flowchart of how you proceed from one screen to the next if you simply enter a carriage return.
^H or H	Displays the list of special editing commands.
^I	Enables you to insert an additional repetitive-fixed screen in front of the current screen. Otherwise, the command ^I has no effect.
^I <number> = or <number> =	Enables you to add a new line to a repetitive screen. The ^I is optional. Only the line number and an equal sign are required.
^CO	Enables you to insert an identical copy of the current screen in front of the present screen. This command can be used only with a repetitive-fixed screen.
^R or R	Redisplays the current screen, showing any changes made. Entering ^R is the same as entering a null carriage return. The default or previously entered responses are displayed until you enter the ^R command (or <CR>) to show the changes you have made to this screen. If you are in a help screen, the command ^R returns you to the last non-help screen you were on.
^N or N	Displays the next logical screen. For example, if you are entering data on a unit-information screen and enter ^N, the first DUIB screen for that driver is displayed. If you enter ^N again, the first screen of the next driver is displayed, and so on. If you enter ^N in the last screen, the ICU returns to command mode.

^S <string>
or S <string>

Searches repetitive-fixed screens of the same logical type for the specified string. When this command is entered, the search begins in the next screen of that logical type and searches all fields with a character range (for example, 1-31 characters). The search continues until a match is found. If no match is found, the cursor remains at its current position and the ICU displays the following message:

```
No next match found
```

The syntax for this command is

^S (or S) <string>

The following example shows how to use the ^S command. Assume you have 20 DUIB screens for the iSBC 214 driver and you want to find the screen that defines the device name as w0. First, you would get to the first "(I214)" screen. Then you would enter

The ICU searches all the iSBC 214 DUIB screens until it finds "w0". It then displays that screen.

1.11.1 Deleting Data on a Repetitive Screen Format

To delete information from a repetitive screen, you must use the ^D <number> command, where <number> is the number of the line to be deleted. After the line is deleted, the remaining lines are renumbered and the screen is displayed again. The ICU does not allow you to delete a line that is not displayed. To replace a line you must first delete the existing line, and then insert the new line.

An example of how to delete data on a repetitive screen follows. Assume the "Prefix" screen is defined as shown below. The cursor is positioned under the word "Enter". If you wish to delete line 6, you would do so as shown here.

```
(PREF)     Prefixes

    Prefix = 1-45 characters
[1] Prefix = :PROG:
[2] Prefix = :UTILS:
[3] Prefix = :SYSTEM:
[4] Prefix = :LANG:
[5] Prefix = :$:
[6] Prefix = :WORK1:
[7] Prefix = :TMP286:
[8] Prefix =

Enter Changes [Number = new value / ^D Number / ? / H ]
:  ^d 6 <CR>
```

After line 6 is deleted, the screen is redisplayed with lines 7 and 8 renumbered to 6 and 7 as shown here.

```
(PREF)     Prefixes

    Prefix = 1- 45 characters
[1] Prefix = :PROG:
[2] Prefix = :UTILS:
[3] Prefix = :SYSTEM:
[4] Prefix = :LANG:
[5] Prefix = :$:
[6] Prefix = :TMP286:
[7] Prefix =

Enter Changes [Number = new value / ^D Number / ? / H ]
```

1.11.2 Inserting Data on a Repetitive Screen Format

To insert a line on a repetitive screen, enter the insert command ^I (optional), the line number, an equal sign (=), and the new value. When the new line number is inserted, the ICU renumbers the remaining lines and displays the screen again. If the number you enter is larger than the actual number of lines in the screen, the ICU inserts the new line as the last line. Assume you want to insert a new prefix on line 6 of the "Prefix" screen displayed previously. You can enter

or

and the screen will be redisplayed with the new values as shown here.

```
(PREF)     Prefixes

    Prefix = 1- 45 characters
[1] Prefix = :PROG: [2] Prefix = :UTILS:
[2] Prefix = :UTILS:
[3] Prefix = :SYSTEM:
[4] Prefix = :LANG:
[5] Prefix = :$:
[6] Prefix = :CONF1:
[7] Prefix = :TMP286:
[8] Prefix =

Enter Changes [Number = new value / ^D Number / ? / H ]:
```

If you are entering numerical data on a repetitive screen such as the "Memory for System" screen, you can enter the data in any order. However, the ICU automatically arranges your data in the proper order and displays it on the screen. For example, if you enter the following three insert commands

```
^I 1 = 2000H, 4000H
^I 2 = 80000H, 90000H
^I 3 = 10000H, 12000H
```

on the "Memory for System" screen (see the *Extended iRMX II Interactive Configuration Utility Reference Manual*), the ICU sorts the data in ascending order and redisplays the lines as follows:

```
1 = 2000H, 4000H
2 = 10000H, 12000H
3 = 80000H, 90000H
```

1.11.3 Deleting a Repetitive-Fixed Screen

The ^D command enables you to delete information for an entire repetitive-fixed screen; you delete the current screen. You can use this command to delete I/O Jobs, User Jobs, OS Extensions, and Remote File Servers, as well as Intel and user devices. If you want to delete a device driver, it is only necessary to delete the Driver screen for that device. The ICU automatically deletes all the Unit and DUIB screens associated with it (see the *Extended iRMX II Interactive Configuration Utility Reference Manual* for more information).

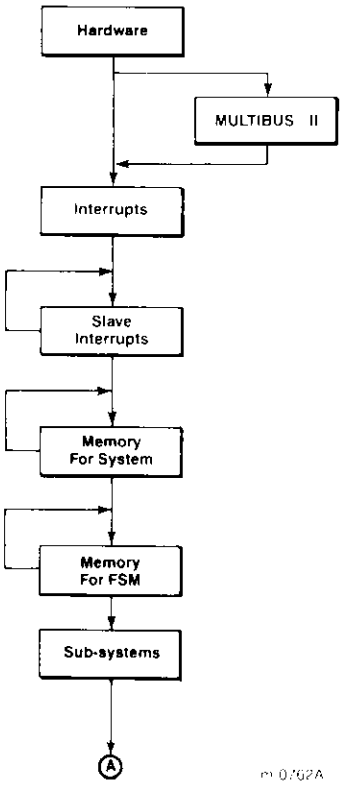


Figure 1-4. ICU Flowchart

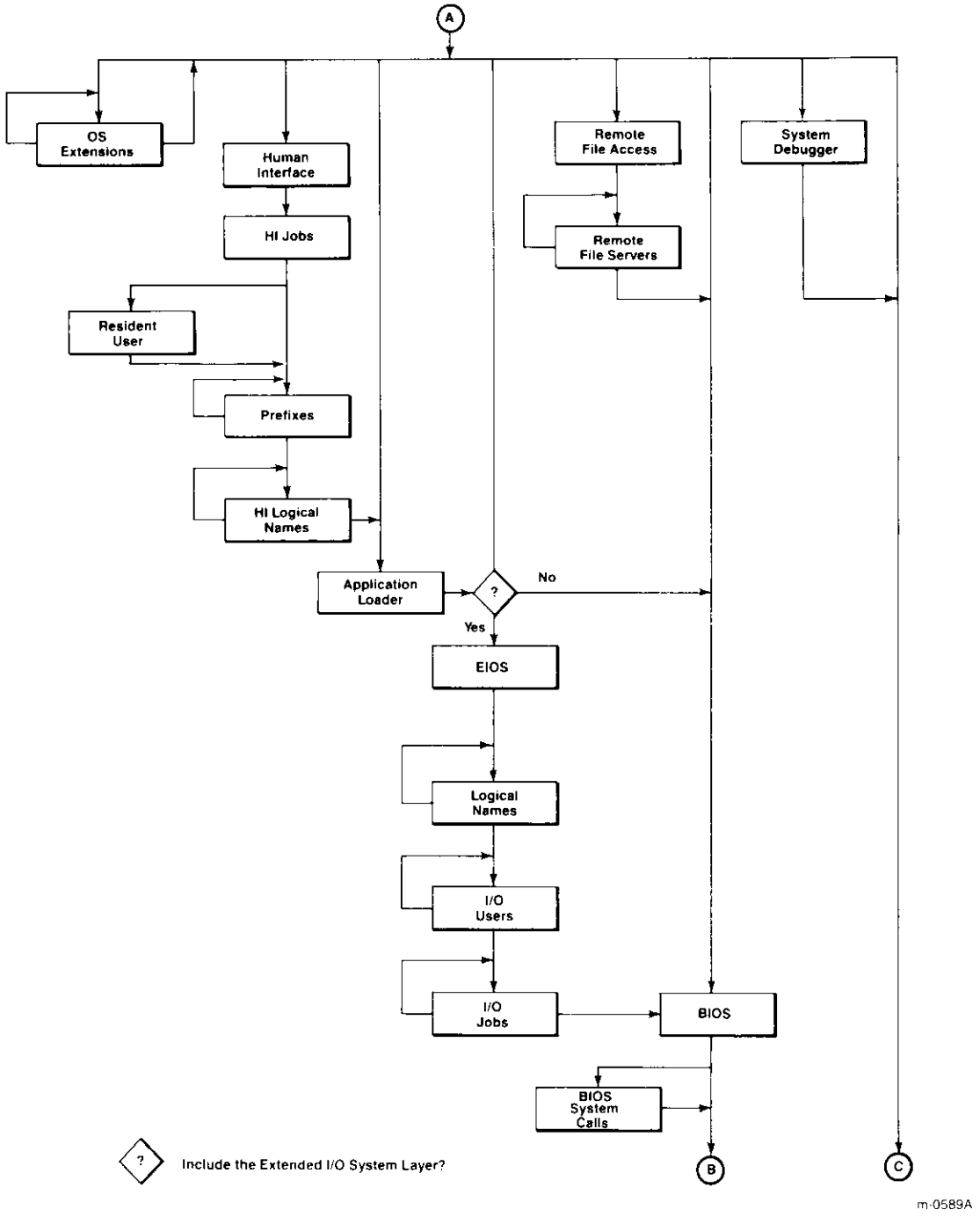


Figure 1-4. ICU Flowchart (Continued)

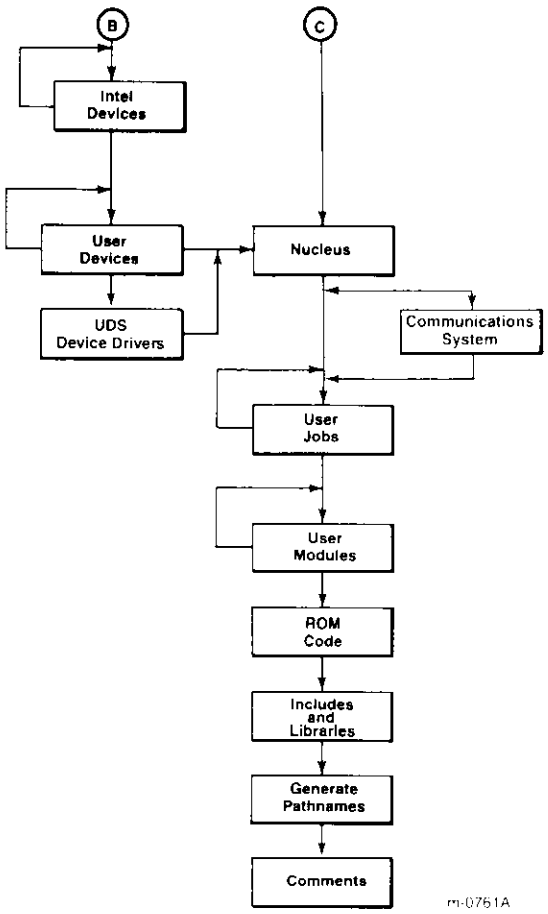


Figure 1-4. ICU Flowchart (Continued)

1.11.4 Inserting a Repetitive-Fixed Screen

The ^I command enables you to insert an additional screen of information between two existing screens. (This can be used only with the repetitive-fixed screen format.) Use this command on the screen you wish to precede. For example, if you have three User Jobs and wish to insert a fourth job between the second and third job, use the ^I command on the screen for the third job.

The copy command (^CO) can also be used to insert an additional repetitive-fixed screen. The copy command inserts a copy of the current screen in front of itself. The only difference between the insert command and the copy command is that the copy command uses the current screen values rather than the default values.

1.12 ICU ERROR MESSAGES

During the interactive portion of the ICU process, two types of error messages can occur:

- interactive error messages
- internal ICU errors

The interactive messages are the most frequently encountered, and are self-explanatory. The ICU internal error messages should not occur. The following sections explain these errors in more detail.

1.12.1 Interactive Error Messages

The ICU accepts data that you enter only if it lies within the range of acceptable values. Usually, the range of acceptable values for a given prompt appears in brackets "[]" on the prompt line. If you specify a value outside the range of acceptable values, the ICU displays one of the following messages, depending on the kind of value it requires (all of these messages are preceded by *** ERROR -):

- number expected or number too large
- number is not within its range
- address expected
- the selector is not within its range
- offset in address is not a number
- string too long
- a prefix of a legal string expected
- "Yes or No" expected
- the field is "Req"; cannot be changed

- erroneous delimiter
- the line entered overlaps

When an error occurs, the ICU does not change the current value of the parameter. If the values you specify lie within the range of acceptable values, the ICU accepts them without checking their reasonableness. Therefore, if you enter values that cause the ICU to generate a nonfunctional version of the operating system, neither the interactive phase nor the generation phase of the ICU will flag these values as errors.

When the ICU leaves the change phase and returns to the initial menu screen, it performs a number of logical tests, such as checking that the memory locations reserved for the system and the Free Space Manager do not overlap. If it detects a logical error, the ICU issues a self-explanatory error message. You must then make the necessary corrections to your definition file or you will not be able to generate a working system.

1.12.2 Internal ICU Errors

If during execution the ICU encounters an internal error such as the Screen Master File or the Template file being corrupted, it displays the following message:

```
*** ICU Internal Error - <number[,s]>
```

where <number[,s]> can be either one number or two numbers separated by a comma. The numbers represent an internal code for the ICU and are not meaningful for the user. Internal ICU errors rarely occur, but if you should receive this error message, follow these guidelines.

1. First, assume your definition file has become corrupted, and try running the ICU again with a new definition file.
2. If Step 1 is not the solution, try running the ICU with a new Screen Master File and a new Template file. Default versions of these files are kept in the directory :CONFIG:default.
3. If neither of the above solve the problem, contact your local Intel sales office.

1.13 UPGRADING DEFINITION FILES

There are three reasons you may have to upgrade definition files.

- To make iRMX II.1 definition files compatible with iRMX II.3.
- To add Intel-supplied changes
- To add user device drivers

To upgrade definition files created by iRMX II.1 of the iRMX II Operating System, use the UPDEF Utility.

To upgrade your iRMX II.3 definition files to include Intel-supplied changes or user devices, invoke the ICU with the definition file you want upgraded as input. iRMX II.3 definition files can have two formats:

- ICU standard format with a specific version number
- Backup format (ASCII) used by different versions

The ICU checks the version numbers (see section, "Invocation Error Messages", earlier in this chapter) and decides how to proceed. If it is possible to upgrade the definition file without restoring the backup information, the ICU prompts

```
Do you want to update the file? y/[n]
```

A response of "Yes" causes the ICU to upgrade the file as you input it. You can then proceed with the ICU as usual.

If the ICU must restore to upgrade the definition file (for example, if the ICU is invoked with a backup file or a definition file whose Intel version number differs from the ICU version number), it invokes the restore process and prompts you as follows:

```
Do you want to restore from the file? y/[n]
```

A response of "No" causes the ICU to stop executing. A "Yes" response means the ICU should restore the backup information contained in the file, and create a new version of the definition file.

If you enter "Yes" and the input file is the same as the output file, you are prompted

```
Enter new output file name:
```

If the output file exists, the ICU displays this message:

```
File <output_file> exists.  OVERWRITE?  y/[n]:
```

While restore is operating, the ICU displays a series of asterisks (*) on the screen. If the restore operation reaches completion with no loss of data, the ICU displays the main menu and you proceed as usual. However, if an error is encountered, the ICU displays the following message and exits.

```
*** ERROR while restoring
The Definition File has been restored to file:  <file-name>.def
Inspect the log file: <file-name>.log
```

The ICU writes the backup information that was not restored to a log-file. The log-file lists each screen name followed by any errors that occurred while restoring that screen. It also lists abbreviations of fields which were not restored. The log-file has the same name as the output file but with a ".log" extension. The log-file makes it easy to compare the backup definition file and the restored file to see which values were not restored. You should then run the ICU correcting the fields in error. After that you can proceed as usual.

Assume that while restoring from file up0.def, the ICU was not able to restore the "CF" parameter on the "Hardware" screen. The log file would look like this:

```
ICU286 <version number> Restoring from file : up0.def  <date>  <time>

----        Screen : HARD       ----
*** ERROR - number expected
                                    In field :  CF

----        Screen : INT        ----
```

The error messages in the log-file are the same as the ICU interactive error messages.

This example shows only a portion of the log-file. However, the actual file lists all the screen names. The version number, date, and time in the heading are variables.

1.14 THE ICUMRG UTILITY

The ICUMRG Utility supplied with the ICU provides the ability to include configuration support for new drivers. The ICUMRG Utility allows you to

- Integrate new Intel device drivers with a previous version of the operating system
- Integrate user-written device drivers into the operating system

The ICUMRG Utility combines the main Screen Master File (ICU286.SCM) and the main Template File for System Generation (ICU286.TPL) with the Screen Master File (SCM) and Template Files (TPL) for the new driver.

If you are adding an Intel-supplied driver, both the SCM and TPL files are supplied with the update package. In addition to adding your device driver, the ICUMRG Utility updates the "Intel Device" screen to include the new device, and changes the help message that lists all the screen names. Upon completion, ICUMRG updates the Update version number.

If you are adding a user-written device, the SCM and TPL files were previously generated by the UDS Utility (see the *Extended iRMX II Device Drivers User's Guide* for more information). Upon completion, ICUMRG updates the User version number.

After running ICUMRG, the version numbers of the new ICU and your definition files are different. To continue using your definition files, invoke the ICU as usual. The ICU will check for version number consistency, and if necessary issue a warning and a prompt (see section "Invocation Error Messages", earlier in this chapter) to which you should respond "Yes". The ICU then updates your definition files and continues executing. Figures 1-5 and 1-6 give the logical flow of the ICUMRG Utility when adding either an Intel device driver or a user device.

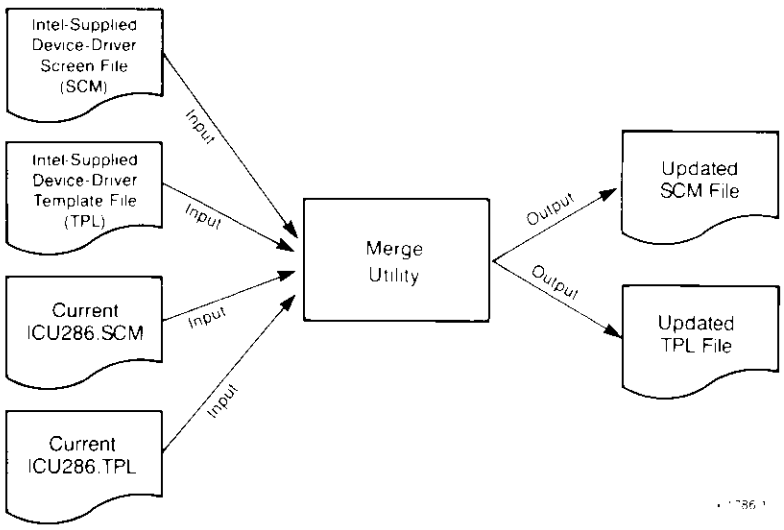


Figure 1-5. Merging Intel Device Drivers

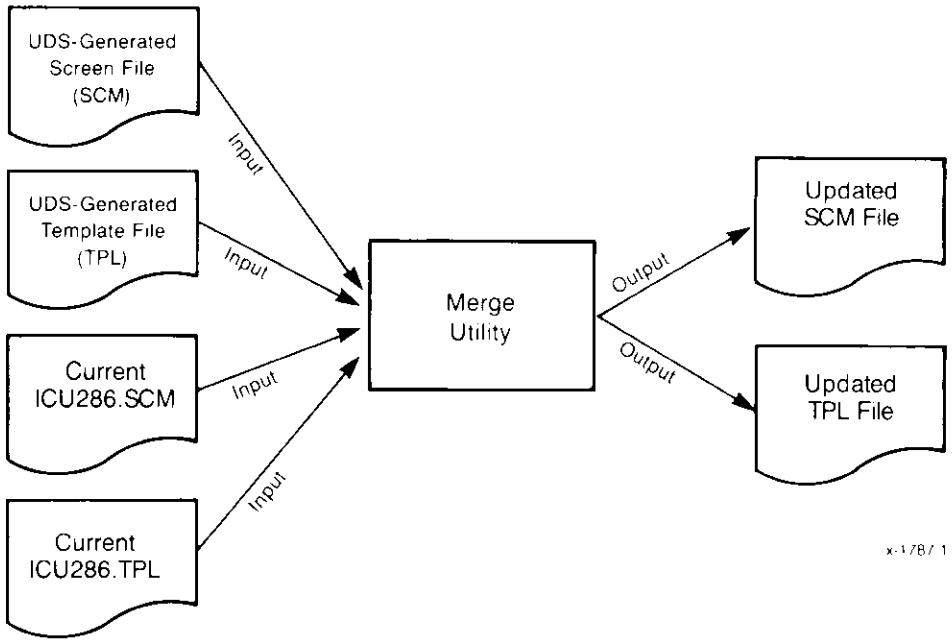


Figure 1-6. Merging User Devices

1.14.1 Invoking ICUMRG

Before invoking ICUMRG be sure that the ICU286.SCM, ICU286.TPL, and ICUMRG files are in the same directory. To invoke the ICUMRG Utility enter

ICUMRG input-file(root) TO newicu-file(root)

where

input-file(root) The input-file name without the extension providing the input to ICUMRG. All extensions included in the pathname are ignored, and replaced by SCM and TPL. The ICUMRG Utility searches the input directory for

input-file.SCM - contains all information about the new driver and the new "Intel Device" screen.

input-file.TPL - contains information needed for generation of new screens.

The ICUMRG Utility also uses ICU286.SCM and ICU286.TPL as input.

newicu-file(root)	The name of the two updated files, without their extensions, created by the ICUMRG Utility. All extensions included in the pathname are ignored, and replaced by SCM and TPL. ICUMRG creates newicu-file.SCM - contains the ICU Screen Master File updated with the new device driver. newicu-file.TPL - contains the ICU Template File updated with the new device driver.

Be aware that the ICUMRG utility always merges your .SCM and .TPL files with the ICU files ICU286.SCM and ICU286.TPL. If you plan to add support for several drivers to the ICU, make sure that the ICU286.SCM and ICU286.TPL files contain the latest version of your merged ICU files. Otherwise, ICUMRG will merge your driver information with outdated ICU files.

NOTE

Before changing the name of any ICUMRG output files to ICU286.SCM and ICU286.TPL, save the original files by copying them to other files (such as ICU286OLD.SCM and ICU286OLD.TPL). Although ICUMRG allows you to add support for new drivers, once you add that support, there is no way to remove it. If the device driver you added contains an error, you must revert back to the original .SCM and .TPL files.

1.14.2 ICUMRG Example

The following example shows how to add a device - D219.

The input files are	ICU286.SCM AND ICU286.TPL (located in same directory as ICUMRG) D219.SCM and D219.TPL
The output files are	ICUNEW.SCM and ICUNEW.TPL

Upon completion the system prompt is displayed. You are then ready to run the ICU and generate your system.

1.14.3 ICUMRG Error Messages

The ICUMRG utility generates an error message if one of the following occurs:

- it is not invoked correctly
- an I/O error occurs
- the version numbers are inconsistent
- either the SCM or TPL files are not valid

Invalid invocation of ICUMRG causes one of the following self-explanatory error messages to be displayed.

- `parameters required`
  `USAGE: ICUMRG infile TO outfile`
- `missing "TO outfile"`
  `USAGE: ICUMRG infile TO outfile`
- `missing "TO"`
  `USAGE: ICUMRG infile TO outfile`
- `missing "outfile"`
  `USAGE: ICUMRG infile TO outfile`
- `too many parameters`
  `USAGE: ICUMRG infile TO outfile`

In addition to the invocation error messages, ICUMRG issues the error messages listed below.

- `*** UDI Error - <exception-code>, <mnemonic>`

  An error was detected by the UDI. The mnemonic explains the cause of the error. For example, you can receive this error message if ICUMRG cannot successfully change the extension.

- `*** Error - input file same as output file`

  The input and output files cannot be the same.

- `*** I/O Error in file: <file name>`
  `<excep-code>, <mnemonic>`

  An I/O error occurred. For example, the ICUMRG utility was not able to create, open, read, write or seek one of the specified files.

- `*** Error - <file name> is not a valid SCM file`

  The data in the SCM file is not valid.

- `*** Error - <file name> is not a valid TPL file`

  The data in the TPL file is not valid.

- `*** Error - inconsistency in the version of the internal ICU files`

  Versions: INTEL UPDATE USER

ICU286.SCM	<Intel> + <Update> <User Version>
ICU286.TPL	<Intel> + <Update> <User Version>

  There is an inconsistency in the version numbers of the ICU286 SCM and ICU286 TPL files.

- `*** Error - inconsistency in the version of the internal ICU files`

  Versions: INTEL UPDATE USER

Input Scm File	<Intel> + <Update> <User Version>
Input Tpl File	<Intel> + <Update> <User Version>

  There is an inconsistency in the version numbers of the input SCM and TPL files.

- `*** Error - <screen-abbr> screen already exists in ICU286.SCM`

  Duplicate screen names are not allowed. You are probably merging the wrong SCM and TPL files, thus causing a duplicate name to be created.

- `*** Error - unexpected end of TPL file <file name>`

  An unexpected end of file in the TPL file was encountered.

CHAPTER 2
GENERATING YOUR SYSTEM

2.1 INTRODUCTION

The process of generating your configured system consists of the following steps:

- Generating configuration files.
- Executing a SUBMIT file that compiles, assembles, binds, and builds all necessary files.

2.2 GENERATING CONFIGURATION FILES

By using the ICU, you can define the operating system that best meets your individual needs. This process takes place while you are editing your definition file. When you have completely defined your system, return to command mode to generate your configured system as follows:

1. Use the List command to create a file that records your system configuration.
2. Use the Generate command to generate your configuration files.
3. Use Exit to save your changes and exit the ICU.

The following screen shows the results of having used the G[enerate] command to generate all the required configuration files (assuming the definition file used was newfile.def).

```
ENTER COMMAND : g

 ENTER a letter to be used as prefix:a

 The prefix letter is: A
 Beginning NUCLEUS File Generation
 ..............................................DONE
 Beginning BIOS File Generation
 ..............................................DONE
 Beginning EIOS File Generation
 ..............................................DONE
 Beginning LOADER File Generation
 ..............................................DONE
 Beginning HI File Generation
 ..............................................DONE
 Beginning UDI File Generation
 ..............................................DONE
 Beginning SDB File Generation
 ..............................................DONE
 Beginning Submit File Generation
 ..............................................DONE
 Beginning Build File Generation
 ..............................................DONE

 NOTE:  To GENERATE your system submit  NEWFILE.CSD

For general help in any screen enter H <cr>

The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND :
```

The files listed in Table 2-1 are the configuration files that define your system. The system processes these files during execution of your SUBMIT file. The ICU creates the SUBMIT file with the same filename as your definition file (with a .CSD extension). For example, the definition file used in the previous screen was labeled NEWFILE.DEF. Therefore, the SUBMIT file is called NEWFILE.CSD.

If you use the prefix option, be sure to choose a unique prefix each time you generate your system. If a file of the same name already exists, the ICU overwrites the old file with the new file.

Table 2-1 shows file names created using no prefix (carriage return only). If you enter any character other than carriage return when prompted for a prefix, that character is added as the prefix to the file names.

Table 2-1. Files Created by the G[enerate] Command

File Name	Screens Used to Define the File
NTABL.A28	Nucleus
NUCDA.A28	Nucleus, Hardware, Interrupts, Slave Interrupts, OS Extensions, ROM Code
NJOBC.A28	OS Extensions, User Jobs
ITABL.A28	BIOS System Calls, Remote File Access
ICDEV.A28 and ITDEV.A28	All Intel and user devices, Remote File Access
ETABL.A28	None
EDEVC.A28	EIOS, Automatic Boot Device, Logical Names
EJOBC.A28	I/O Users, I/O Jobs
HCONF.P28	Human Interface, HI Jobs, Resident User, Prefixes, HI Logical Names
LTABL.A28	None
LCONF.P28	Application Loader
SDBCN.A28	System Debugger
UTABL.A28	None
NROMC.A28	Hard, MBII, Mems, ROM

CAUTION

Changes made to the ICU definition file are not reflected in your configuration files until you generate.

2.3 EXECUTING THE SUBMIT FILE

After you exit the ICU, execute the SUBMIT file and wait for your system to be generated.

The SUBMIT file assembles or compiles any configuration files generated by the ICU and binds the object files with any needed libraries used by a subsystem. It then builds the system. The syntax for invoking the SUBMIT file is

```
SUBMIT output-file[.CSD] [to filename] [echo]
```

where:

output-file	The name of your definition file.
filename	A file that the system creates to contain the output of the SUBMIT command.
e[cho]	Sends a copy of the data read to the screen.

For more information on the SUBMIT command, see the *Operator's Guide to the Extended iRMX II Human Interface*.

2.3.1 Assembling the Configuration Files

The SUBMIT file generated by the ICU identifies the configuration files that must be assembled or compiled for each of your subsystems. The number of files assembled varies from system to system and depends upon the features that you choose. No errors should be encountered during this phase. Figure B-36, in Appendix B, gives an example of the SUBMIT file output during this phase of the configuration process.

2.3.2 Binding the Individual Subsystems

As soon as ASM286 generates the object files for a given subsystem, the SUBMIT file initiates BND286 to bind these object files together with any libraries needed by the subsystem. Any warnings generated during this phase should be ignored. Explanations of the various warnings appear at the end of this section. Figure B-36, in Appendix B, shows some of the output generated during this phase of the configuration process.

2.3.3 Warning Messages

When you invoke the system generation SUBMIT file, a number of warning messages may be issued by BND286. These are normal messages and are not critical.

- `WARNING 151: UNRESOLVED EXTERNAL SYMBOLS`

  This warning is the most common. It indicates that BND286 did not resolve all the external symbols declared. You may ignore this warning since the missing symbols are resolved only at build time. It may appear when binding the Nucleus or after the first phase of binding either the Basic I/O System or the Extended I/O System. It may also appear after the second phase of the EIOS, if you are binding a first-level I/O job.

- `WARNING 133: SEGMENT LIMIT DECREASED DUE TO SEGSIZE VALUE`

  This warning is expected in the Nucleus. It indicates that the size of the segment named is being decreased because of a SEGSIZE specification.

2.3.4 Building the System

After the SUBMIT file has completed the assembling and binding of each of the subsystems, it builds the system by invoking the BLD286 utility. The invocation for BLD286 is contained within the SUBMIT file. An example of this can be seen in Figure B-36.

Entering the Generate command produces a build file with the same name as the definition file but with the extension BLD. This serves as the input file to BLD286. The output file created by BLD286 is the file you define in the "Generate File Name" screen (see the *Extended iRMX II Interactive Configuration Utility Reference Manual*). No errors should occur during this phase of the ICU. However, you can expect one warning which can be ignored. The warning is

```
***WARNING 269:  <line-number>, NEAR 'CLI_DATA', SEGMENT SIZE REDUCED
```

This warning appears after the following line of code in the build file:

```
cli_code(dpl=0), cli_data(limit=0,dpl=0)
```

2.3.5 Error Messages

In addition to warning messages, the language utilities can return error messages. Error messages are not normal and you should not ignore them. They indicate serious problems that prevent the successful generation of your system. The following error messages can appear.

- `0021: E$FILE_NOT_EXIST`
  `8042: E$NOT_CONNECTION, command aborted by EH`

  One of these messages might appear if you enter an invalid pathname as input to the ICU.

- `0026: E$FILE_ACCESS`

  This message may appear if there is no read access to the input file, no add entry access to the directory, or no write access to the output file.

- `ERROR 118: INPUT SEGMENTS EXCEED TARGET MEMORY`

  The memory blocks you declared in the "Memory for System" screen are not large enough for the system you defined. In this case, enter the ICU again, increase the system memory, and decrease the Free Space Manager memory. This error message may result from installing an update which increases the size of the operating system.

CHAPTER 3
PREPARING APPLICATION JOBS

3.1 INTRODUCTION

Once you have prepared your application jobs, you should locate your first system in RAM to facilitate testing and debugging of your programs. It is much easier to test and debug your programs in RAM than it is to reburn your PROM devices when you detect errors. After debugging in RAM, you can locate the final system in PROM/RAM or copy it to a secondary storage device and load it with the Bootstrap Loader.

Putting together a RAM-based system consists of the following steps:

1. Using the ICU to define your system
2. Preparing your application code
3. Compiling and binding the application jobs
4. Using the ICU to generate the system with your application (not needed if the Application Loader loads your application)
5. Loading and testing the system

This chapter describes how to prepare your application code, compile and bind it, and build a system with your application jobs (steps 2 through 4 above). Both loading and testing your system are described in Chapter 5 of this manual.

3.2 PREPARING APPLICATION CODE

You can write the code for your application tasks in any language supported by the iRMX II Operating System. This manual assumes that you are using PL/M-286. In order to use assembly language, you must adhere to the PL/M-286 calling conventions described in the *ASM286 Macro Assembler Operating Instructions* manual. The *Extended iRMX II Programming Techniques* manual also contains information to help you write your application code, especially assembly language applications.

When writing your application code there are additional instructions you should follow in order to use all the features of the iRMX II Operating System. The following sections provide this information.

3.2.1 Language Requirements

Adhere to the following language requirements when writing your task code:

- Make certain any utilities you use are linked to the Extended iRMX II UDI libraries.
- In general, you should designate all of your tasks as procedures. Designation of initial tasks is the only exception to this recommendation. Refer to *Extended iRMX II Application Loader User's Guide* for details about main modules and procedures.
- If you are compiling your PL/M-286 code using any model other than LARGE, specify the ROM compiler control. This causes the compiler to place the CONST segment in the CODE class, where it can be more easily loaded into PROM. You do not need to specify the ROM control for those programs compiled using the LARGE model. The compiler does this automatically for the LARGE model.
- Use the DATA and INITIAL PL/M-286 statements with care. The DATA statement is valid only if you use the PL/M-286 LARGE model of segmentation or if you specify the ROM compiler control. The INITIAL statement cannot be used in a procedure if you put that procedure in PROM. It can be used, however, if you use the Bootstrap Loader or Application Loader to load the procedure into memory.

3.2.2 Include Files

A number of files must be present on your microcomputer system to compile your application software and to configure your operating system. The "Includes and Libraries" screen, discussed in the *Extended iRMX II Interactive Configuration Utility Reference Manual*, selects files that must be present to configure your operating system. This section discusses the files needed to compile your application software.

Any program containing iRMX II system calls must include an external declaration of the system calls. The iRMX II Operating System provides the declaration of the system calls for FORTRAN, PASCAL, and PL/M-286 in files called INCLUDE files. When you install the system as described in the *Extended iRMX II Hardware and Software Installation Guide*, these files are located in directory /RMX286/INC. FORTRAN system calls are in file RMXFTN.EXT, PASCAL system calls are in RMXPAS.EXT, and PL/M-286 system calls are in RMXPLM.EXT.

However, if you are programming in PL/M and your system does not include all the subsystems, or if you are trying to save memory, you may want to use a PL/M INCLUDE file that contains system calls only for a particular layer. These files are

Subsystem	File Name
Nucleus	NUCLUS.EXT
BIOS	BIOS.EXT
EIOS	EIOS.EXT
Application Loader	LOADER.EXT
Human Interface	HI.EXT
UDI	UDI.EXT

3.3 DETERMINING MEMORY LOCATIONS

An iRMX II system must be located either entirely in ROM or entirely in RAM. Intel recommends that you put your system in RAM until you have completed the testing and debugging stages. To determine your application's memory requirements, you should add the size of your application code to the amount of memory required by the system. After calculating your system's memory requirements, you must determine its physical location in memory and enter the starting and ending addresses on the "Memory for System" screen.

Some additional factors should be taken into consideration when determining the physical address of your system. All systems have a minimum address at which they can start depending on the elements comprising the system. All systems must take the following memory locations into consideration:

- 0B8000H-0BFFFFH, default addresses required by the second and third stages of the Bootstrap Loader.
- the top 32K bytes, if the system includes the iSDM monitor and the first stage of the Bootstrap Loader (the top 256K bytes, if the system is a System 300 Series Microcomputer).

In addition, if your system includes either a RAM disk driver or a communication board, you must be careful not to include the board's dual-port memory in the memory you declare for your system. For example, if your system includes an MSC driver, you must reserve 68 bytes of memory in the lower megabyte (in an Intel-supplied system, the first 2000H bytes are reserved for such data). After you have reserved room for all the data requiring fixed locations, you can locate your application anywhere within the 16M byte memory of the iRMX II Operating System. For more information on calculating the exact memory locations, see Chapter 2 of the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

Use the memory screens to define the memory for the system and the Free Space Manager. Intel recommends that first you pad the memory locations to leave room for any device drivers that may be added or changes that may be made during development. After running BLD286, you can reduce the memory to the actual size necessary by re-invoking the ICU and editing the memory screens (see section "Minimizing the Memory Address Space", later in this chapter).

3.4 BINDING AND BUILDING YOUR APPLICATION JOBS

Application jobs are included in the system by using BND286 and BLD286. You must bind each application job with its offspring jobs and the interface libraries discussed later in this section. The following sections describe the binding and building process, and the interface libraries in more detail.

3.4.1 BND286

The BND286 command is used as shown below to bind your first-level application jobs. This command is described in detail in the *iAPX 286 Utilities User's Guide*. The following invocation of BND286 applies to both the iRMX II and Series IV systems.

```
BND286                              &
    app_job.obj,                    &
    interface.lib                   &
OBJECT(app_job.lnk)  NOLOAD         &
NOPUBLICS EXCEPT(start_address, public_variable, exc_handler_address)
```

where:

app_job.obj	Pathname of the file containing the object code for your application job. You do not need to provide this code in one file; you can bind in several files or libraries at this point.
interface.lib	Pathname of the file containing the interface libraries for the system calls included in your jobs. These interface libraries are described in later paragraphs of this section.
app_job.lnk	Pathname of the file in which BND286 places the module containing your bound application code. Use this file as the input file when configuring your application job on the "User Modules" screen.
start_address	The starting address of your application. The Nucleus uses this name to obtain the selector and offset of the initial task. This must be the same name you entered as the Task Start Address on the "User Jobs" screen.

exc_handler_address — The starting address of your exception handler. You must enter this value if the starting address is not zero. The Nucleus uses this name to obtain the selector and offset of the exception handler. This must be the same name you entered as the Exception Handler Start Address on the "User Jobs" screen.

public_variable — The PUBLIC name of a variable in your data segment. The Nucleus uses this name to obtain the data selector number. This must be the same name you entered as the Public Variable Name on the "User Jobs" screen. If there is no data segment or the application initializes its own data segment, it is not necessary to specify this parameter.

You should be aware of the following requirements when binding an application job.

- Use the NOLOAD option of BND286. This causes the System Builder (BLD286) to combine your application with the system to create a bootloadable file.
- Ensure that the output of BND286 includes three PUBLIC names, one for the start address, one for a variable in the initial data segment of your application job, and one for the exception handler (the data segment and exception handler are optional). These names must be same as the names you specified on the "User Jobs" screen. These public names allow the system to create a job and start its execution at the correct address with the corresponding data segment intact.
- Bind the appropriate interface libraries to your application job, if you use any iRMX286 system calls. The interface libraries contain routines that satisfy external references to system calls. The name of the library that you must bind in with your application code depends on which model of PL/M-286 segmentation the jobs were compiled under. Table 3-1 shows the correlation between models of segmentation and interface libraries. Specify these libraries as the last modules in the BND286 input list so that they can satisfy references from all bound modules. Notice that no library exists for the SMALL model of PL/M-286 segmentation; except for Universal Development Interface (UDI) level applications, the iRMX II Operating System does not support applications compiled in SMALL.

Table 3-1. Interface Libraries as a Function of PL/M-286 Models

Subsystem	SMALL	COMPACT	LARGE or MEDIUM
All subsystems, except UDI		RMXIFC.LIB	RMXIFL.LIB
UDI	UDIIFS.LIB	UDIIFC.LIB	UDIIFL.LIB

After you have bound your object code using BND286 with the NOLOAD option, you should

1. Invoke the ICU. You may want to use the Level of Detail option and select Jobs from the displayed menu to see all the job screens that require changing.
2. Enter the application job onto the "User Job" screen.
3. Enter the pathname of the object file, created by compiling and binding the code, onto the "User Module" screen.
4. Ensure that you have enough memory for the system, including your application jobs.
5. Generate the system by entering the "G" command and submitting the output-file.CSD. (Invoking the ICU system generation submit file activates BLD286.)

The iRMX II Operating System does not allow you to build your application system in separate bootload files. With BLD286 you build the system as a unit. Figure 3-1 illustrates the bind and build procedure.

3.4.2 Minimizing the Memory Address Size

When you originally located your application jobs, you may have included extra memory to accommodate changes during development. However, in the final system, after binding and building, you can eliminate some of the extra space. BLD286 creates a map file with the same name as the system, but with the extension MP2, as part of its output. By looking at the MP2 file, you can find the exact starting addresses of the descriptor tables, the monitor, and all the other constants in your system. To tighten up memory, change the system memory definition and generate your system again using the ICU.

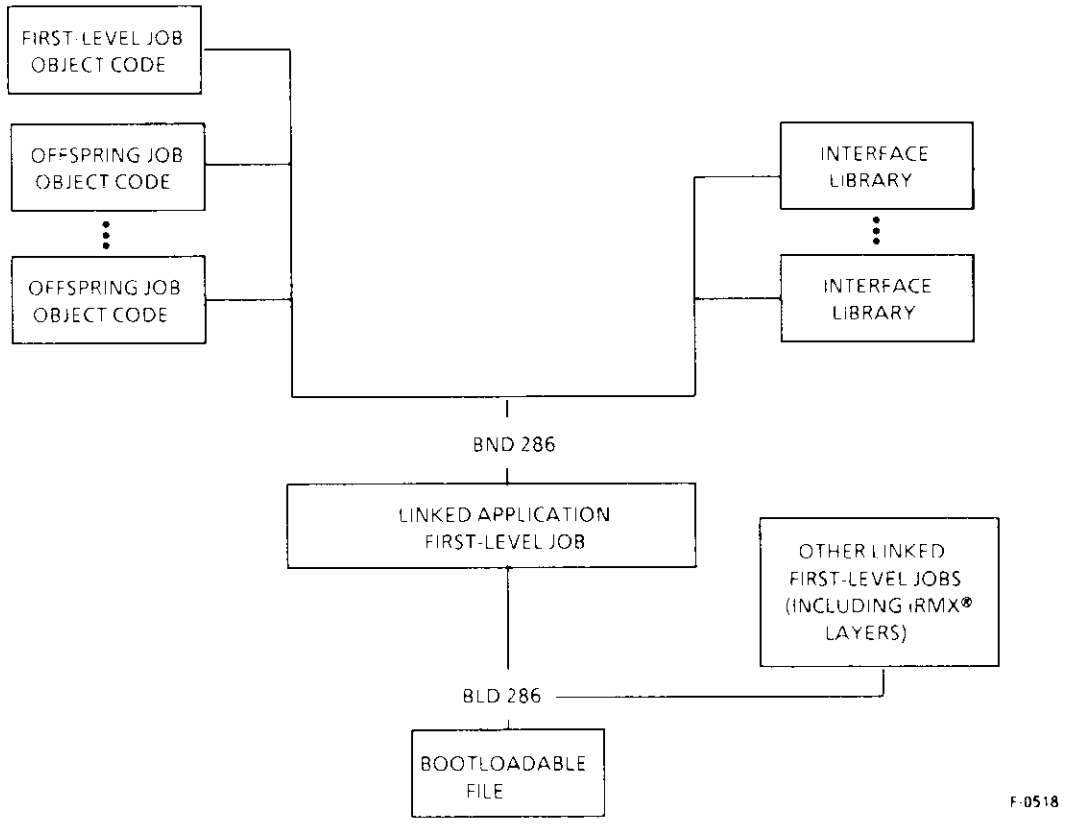


Figure 3-1. Application Job Bind and Build Procedure

3.4.3 Building a ROM-Based System

Before you burn your system into ROM (PROM) devices, you should first be confident that your code is fully debugged. You must also know the size of all the code and data segments. In a RAM-based system, the amount of memory needed by the code and data segments is reserved for the system on the "Memory for System" screen. The remaining memory is available for the Free Space Manager. However, in a ROM-based system, it is necessary to copy the Global Descriptor Table (GDT), Local Descriptor Table (LDT), Interrupt Descriptor Table (IDT) and all writeable segments into RAM. Therefore, you should subtract the amount of memory required by the descriptor tables and the writeable segments from the memory available for the Free Space Manager. The actual area to be used for the system RAM is defined by the "RAM Start Address" parameter and the size of the RAM segments. Ensure that this area is not reserved for the system or the Free Space Manager.

To determine the size of your code and data segments, follow these guidelines.

1. Bind your application using BND286.
2. Run the ICU to create the generation SUBMIT file.
3. Invoke the SUBMIT file.
4. Read the memory map (.MP2 file) created during the build phase to find the start address for the GDT in ROM, the amount of ROM your system uses, and the amount of RAM your system requires. Calculate the RAM memory required by adding the size of the IDT, the final GDT (the number of GDT entries multiplied by 8), the final LDT, which is the same as the final GDT, and the sum of all the data segments with the WRITEABLE attribute. To this number add 2700 bytes which are used as a work area during system startup.
5. Invoke the ICU and remove the size of the memory calculated in step 4 from the memory defined on the "Memory for System" screen.
6. Rerun the ICU to generate the final SUBMIT file.
7. Invoke this system SUBMIT file.

The final file generated after rerunning the ICU with the correct parameters does not include the start address of your system, that is the initial JMP at address 0FFFFF0H. You must burn that into ROM separately. If you specified that the initialization code resides somewhere in the top 64K bytes (not below address 0FF0000H), then you can burn a short jump to that address. Otherwise, you need a FAR jump, which may cause a problem. The 80286 processor resets the high 4 address bits. This means you can only jump to the lowest megabyte, but your ROM is usually in the high megabyte. Boards such as the iSBC 286/12, iSBC 386/2X, and iSBC 386/3X solve this problem by setting the 4 address bits to one until a specific OUT command is issued. This allows you to perform a FAR jump to anywhere in the high megabyte.

The initialization routine resides in the segment NUCDAT.CODE_ROM, and its entry point is at offset 12H. To obtain the entry point address, add the entry routine offset (12H) to the address of NUCDAT.CODE_ROM (which you specified during ICU configuration). For example, if the address of NUCDAT.CODE_ROM is 0FC0000H, the address of the ROM-initialization routine would be 0FC0012H. At location 0FFFFF0H in your PROMs burn a FAR jump to the address you derived for the ROM-initialization routine.

Appendix C provides an example of how to place your code into PROM. It lists both the hardware and software necessary to do this. Refer to this appendix for further information.

CHAPTER 4
ADDING USERS TO YOUR SYSTEM

4.1 INTRODUCTION

To function correctly, a system configured with the Human Interface requires information about all users (operators) and terminals that intend to access the system via the Human Interface. Two types of users exist for your system: a resident user and non-resident users.

4.2 THE RESIDENT USER

The resident user becomes part of your final system and resides in memory along with the rest of the Operating System (thus, the term "resident user"). Two types of resident users exist: a recovery resident user and a non-recovery resident user. The recovery resident user gains control only if an initialization error occurs during initialization of the Human Interface. Regardless of the type, the resident user occupies one of the system terminals and is created before non-resident users. The Operating System can contain information about only one resident user.

Including a resident user type in your system is called resident user configuration. Resident user configuration is accomplished by supplying information to the Human Interface (HI) screen during ICU configuration of the Human Interface. Refer to the *Extended iRMX II Interactive Configuration Reference Manual* for detailed information needed for resident user configuration.

4.3 NON-RESIDENT USERS

Non-resident users are users that can access the system using the Human Interface logon procedure. If your system is to be a multiple-user system, you need to define to the Human Interface all the non-resident users that can access the system. Configuration for non-resident users occurs through the Human Interface PASSWORD command and possible editing of several user definition files. These files define user names, limitations, passwords, terminals, and terminal characteristics to the system.

The process of adding non-resident users to your system is called non-resident user configuration. The files involved are called non-resident configuration files.

The system manager (who has user ID 0) can modify these files to add users or terminals, delete users or terminals, or change characteristics of users or terminals. Depending on the type of modifications made, the changes take effect either the next time the affected user logs onto the system or the next time the system is initialized. To prevent unauthorized users from changing the system configuration, the system manager should be the only user with change access to these files.

Refer to the *Operator's Guide to the Extended iRMX II Human Interface* for detailed information on non-resident user configuration.

CHAPTER 5
LOADING AND TESTING THE SYSTEM

5.1 INTRODUCTION

After you run the SUBMIT file generated by the ICU, you are ready to load the system into RAM and test it. The system RAM code is contained in the file that you specified while running the ICU. There are several different ways in which you can load your system into RAM.

5.2 LOADING YOUR SYSTEM INTO RAM

If you are using a Series IV development system, use the iSDM System Debug Monitor to load your system from disk into RAM. The iSDM monitor is described in the *iSDM System Debug Monitor Reference Manual.* Should you have a system that uses D-MON386, you load the system into RAM using the D-MON386 B command (Bootstrap Loader command).

If you are using your System 300 Series Microcomputer or a MULTIBUS II system as a development tool, use the Bootstrap Loader to load your system into RAM. The procedures for using the Bootstrap Loader are described in the *Extended iRMX II Bootstrap Loader Reference Manual.*

5.3 INITIALIZING YOUR SYSTEM

After you load your system, you must initialize it. If you are using the Bootstrap Loader this process takes place automatically. If you did not load your system using the Bootstrap Loader, refer to the appropriate manual for instructions on how to initialize your system by starting execution from the beginning of the Root Job.

5.3.1 Initialization

An iRMX II Operating System can be configured to include your own code as a first-level job or as a first-level I/O job. When created, such a job contains only a single task. That single task creates or starts the creation of all other objects required by the first-level job. Thus it is referred to as the initialization task for its job, even though it may perform other functions as well. You should synchronize the operation of each initialization task with that of the root task to ensure proper functioning of your application system.

The root task is structured so that it creates the first-level jobs one at a time. It contains a programming loop that in general performs the following:

```
Repeat for each first-level job
    Create first-level job
    Suspend root task (until resumed by a first-level job)
Until finished
End
```

Each time the root task creates a first-level job, the root task suspends itself to allow the initialization task in the new job to perform synchronous initialization. Synchronous initialization consists of functions that must be performed immediately, before some other first-level job is created. Typically, this requires creating objects or making resources available that tasks in first-level jobs, not yet created, expect to be available when they themselves are created. (For example, the initialization task in the Extended I/O System job must create the entire Extended I/O System before it can allow the root task to create other first-level jobs that might make use of Extended I/O System functions.)

When the initialization task finishes its synchronous initialization, it must inform the root task that it is finished, so that the root task can resume execution and create another first-level job. The initialization task must always inform the root task that it has completed its synchronous initialization process by making the following procedure call:

```
CALL RQ$END$INIT$TASK;
```

This procedure call requires no parameters. When you call this procedure, the root task resumes execution, allowing it to create the next first-level job. You must include a call to RQENDINIT$TASK in the initialization task of each of your first-level jobs, even if the jobs require no synchronous initialization. If one of the first-level tasks does not include this call, the root job remains suspended and cannot create any of the remaining first-level jobs.

The amount of synchronous initialization that an initialization task must do depends on your job structure. You may require some of your initialization tasks to create all of the offspring jobs and a number of other objects before calling RQENDINIT$TASK. Some others may have to perform only one or two functions, call RQENDINIT$TASK, and then resume the process of initialization asynchronously. Still other initialization tasks may not have any synchronous initialization requirements and so can call RQENDINIT$TASK before performing any initialization. You must determine how the pieces of your system interact, and how they must be synchronized.

Another important factor in initialization is the order in which the root job creates the first-level jobs (see Table 5-1). The amount of processing your initialization tasks must do before calling RQENDINIT$TASK may depend on which jobs the root task has already created and which jobs it has yet to create. The order in which the root task creates first-level jobs depends on the order that you specify these jobs while running the ICU, not on the priority of the tasks in those jobs.

You should always use RQENDINIT$TASK as described in this section in order to perform your synchronous initialization. Otherwise, the root task cannot be resumed and thus, it cannot complete system initialization in the correct order.

Table 5-1. Order of Initialization

Order	Root Job	First-Level Job	I/O User Job
1	Root Job		
2		System Debugger	
3		Basic I/O System	
4		Extended I/O System	
5			I/O User Jobs
6		User Jobs	
7		Human Interface	

5.3.2 System Initialization Errors

If the system encounters an error during the initialization process, it places diagnostic information in the processor registers and halts the processor. If the "Report Initialization Errors" entry on the Nucleus screen is "yes" and your processor board contains the iSDM monitor, a hexadecimal code and a mnemonic are displayed at the console indicating the layer that contains the initialization error. On encountering an initialization error, the subsystem containing the error returns control to the iSDM monitor after writing a message with the following format:

```
<subsystem> Initialization Error: <error code number>
```

This initialization error reporting is selected either for all subsystems or for none of the subsystems. If "Report Initialization Errors" is not configured into the system, the exception code returned by the unsuccessful system call is placed in both the AX register and the first WORD of the Nucleus data segment, NUCDAT. A code indicating the layer that failed initialization is placed in the second WORD of the Nucleus data segment. The system then goes into a infinite error loop. (The codes for the various layers are 1 = Nucleus, 2 = BIOS, 3 = EIOS, 4 = Human Interface.)

The only subsystem that handles an initialization error slightly different is the Human Interface. In addition to the initialization error described above, the Human Interface may issue the following warning if it does not have enough memory to fill the user's request.

```
*** WARNING:  THE SYSTEM DID NOT HAVE YOUR MINIMUM MEMORY REQUIREMENTS
              YOU WILL COME UP WITH ALL THE MEMORY THAT IS AVAILABLE IN THE
              SYSTEM, CONTACT THE SYSTEM MANAGER.
```

In such a case, the user is assigned whatever memory is available at the time.

5.3.3 Completing Initialization

Once initialization is complete, users can create and attach files on the devices specified with the ICU. If the devices are off-line, an exceptional condition code is returned. If one of these devices is switched from on-line to off-line, the Extended I/O System automatically detaches the device, and all file connections on that device are marked invalid by the BIOS. When the unit is switched back on-line, the Extended I/O System automatically attaches it the first time a user tries to create or attach a file on the device. The Extended I/O System performs this service only for devices that it attaches.

5.4 TESTING YOUR SYSTEM

The normal development cycle is to load your system, test it and correct any errors, then reassemble or recompile any appropriate program code. Next, redefine and regenerate your system using the ICU, and load the system again. You can continue this procedure until you have created your target system. Once you have created your final system, minimize the memory locations allocated for the system by editing the "Memory for System" screen (see Chapter 3). You can then copy your final system to PROM or use the Bootstrap Loader to load it from secondary storage.

If you are going to use the Bootstrap Loader to load your system, refer to *Extended iRMX II Bootstrap Loader Reference Manual* for configuration information.

5.4.1 Using the Debugging Tools

The development of every system requires debugging and testing. To aid you in the development of iRMX II-based application systems, the I^2ICE In-Circuit Emulator, the iRMX II System Debugger with either the iSDM System Debug Monitor or the D-MON386 Monitor are available from Intel. The System Debugger extends the capabilities of the iSDM Monitor and the D-MON386 Monitor. In addition to the system debugging tools, Intel has Soft-Scope 286/ to debug user programs loaded by the Application Loader. The following sections describe the advantages of these debugging tools.

5.4.1.1 Advantages of the iRMX® II System Debugger

You can extend the capabilities of the iSDM Monitor or the D-MON386 Monitor by including the System Debugger as part of your operating system. In addition to retaining the features of the monitors, the System Debugger

- Identifies and interprets iRMX II system calls.
- Displays iRMX II objects.
- Allows the user to examine the stack of a task to determine which iRMX II system calls it has made recently.

5.4.1.2 Advantages of Soft-Scope® 286

Soft-Scope is an interactive, source-level debugging tool designed to aid in the debugging of user programs loaded by the Application Loader. It provides the following debugging features:

- Source code interface and on-line listings
- Access to program variables, including arrays and structures
- High-level breakpoints
- Access to assembly level debugging
- iRMX II multitasking support
- iRMX II exception handling
- Access to iRMX II objects such as mailboxes and tasks
- Ability to suspend and resume tasks

5.4.1.3 Advantages of the I²ICE™ In-Circuit Emulator

The I²ICE emulator provides in-circuit emulation for 80286 and 80386 microprocessor-based systems, meaning that it "stands in" for the these microprocessors in your target iRMX II- or Distributed iRMX III-based system. The in-circuit 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.
- Protect memory areas from being altered and trap on attempted access.

5.4.2 Debugging Application Jobs

While you are creating your application jobs, you will probably use the following iterative procedure to remove bugs from your code:

1. Configure your system.
2. Generate your system using ICU generated command files.
3. Test the system to find bugs.
4. If any bugs are found, modify the application code to eliminate the bugs and go to Step 2.

Once you have performed the entire configuration process, you are ready to load the system.

APPENDIX A
FILES CREATED BY THE ICU

A.1 INTRODUCTION

The files listed in this appendix are created/recreated by the ICU or as output of the SUBMIT file.

A.2 CREATED FILES

The table below lists the files that are created/recreated whenever you issue the G command from the ICU and/or invoke the ICU-generated SUBMIT file. The file names listed here do not include the prefix letter that may be added before generation.

Table A-1. Files Created by the ICU and SUBMIT File

Subsystem	Created by ICU	Created by SUBMIT File
Nucleus	NTABL.A28	NTABL.OBJ NTABL.LST
	NUCDA.A28	NUCDA.OBJ NUCDA.LST
	NJOBC.A28	NJOBC.OBJ NJOBC.LST
	NROMC.A28	NROMC.OBJ NROMC.LST
		NUC1.LNK NUC1.MP1 NUCLS.LNK NUCLS.MP1
Basic I/O System	ICDEV.A28	ICDEV.OBJ ICDEV.LST
	ITABL.A28	ITABL.OBJ ITABL.LST
	ITDEV.A28	ITDEV.OBJ ITDEV.LST
		IOS1.LNK IOS1.MP1 IOS.LNK IOS.MP1

Table A-1. Files Created By the ICU and SUBMIT File (continued)

Subsystem	Created by ICU	Created by SUBMIT File
Extended I/O System	EDEVC.A28	EDEVC.OBJ EDEVC.LST
	ETABL.A28	ETABL.OBJ ETABL.LST
	EJOBC.A28	EJOBC.OBJ EJOBC.LST
		EIOS1.MP1 EIOS.LNK EIOS.MP1 EIOS1.LNK
Application Loader	LTABL.A28	LTABL.OBJ LTABL.LST
	LCONF.P28	LCONF.OBJ LCONF.LST
		LOADR.LNK LOADR.MP1
Human Interface	HCONF.P28	HCONF.OBJ HCONF.LST
		HI.LNK HI.MP1
		CLI.LNK CLI.MP1
UDI	UTABL.A28	UTABL.OBJ UTABL.LST
		UDI.LNK UDI.MP1
System Debugger	SDBCN.A28	SDBCN.OBJ SDBCN.LST
		SDB.LNK SDB.MP1
Others	<output-file>.CSD <output-file>.BLD	boot-loadable iRMX II file <bootloadable-file>.MP2

APPENDIX B
EXAMPLE SYSTEM CONFIGURATION

B.1 INTRODUCTION

This appendix contains an example illustrating how to use the ICU to modify an Intel-supplied definition file. This example contains the following descriptions:

- The configuration defined by the Intel-supplied definition file (28612.def).
- The target system, focusing on the differences between it and the supplied configuration.
- The ICU changes required to convert the existing definition file to one corresponding to the target system.

B.2 THE INTEL-SUPPLIED DEFINITION FILE

The existing definition file, named 28612.DEF, defines the 80286-based multi-user system. This particular system configuration has the following characteristics:

- The CPU board is an iSBC 286/10(A) or an iSBC 286/12 board.
- Interrupt levels are assigned as follows:

  Level 0 - System Clock

  Level 1 - System Debugger

  Level 2 - Available

  Level 3 - Used by the Terminal Communications Controller

  Level 4 - Available

  Level 5 - An MSC controller

  Level 6 - An 8274 Terminal Driver

  Level 7 - An 8259A slave PIC

- Up to 8M bytes of RAM, at addresses 2000H through 07FFFFFH. Of these, addresses 05A000H through 07FFFFFH are allocated to the Free Space Manager.
- The system device is :SD:.
- The supplied, ready-to-bootstrap-load file is /BOOT/28612.286.

B.3 DIFFERENCES BETWEEN THE TARGET AND START-UP SYSTEMS

The differences between the target system and the 80286-based multi-user system dictate how you will use the ICU to alter the definition file. These systems differ in the following ways:

- Change the reserved memory address of the MSC driver to prevent an address conflict with the iSBC 220 driver.
- An iSBC 220 SMD controller for an 87M byte disk is an addition to the 80286-based multi-user system. This controller uses interrupt level 2, I/O address 120H.
- An iSBC 208 flexible disk controller added to support 8-inch flexible disks. As our target system will not include iRMX-NET, the iSBC 208 will be placed on interrupt level four and will use slave I/O address 180H.
- A RAM driver is an addition to the 80286-based multi-user system.
- The target system defines 4M bytes of RAM, 3M bytes are used by the system and the Free Space Manager and 1M byte is used by the RAM disk.
- The target system resides in a bootloadable file named /BOOT/SAM286.286.

The name of the new definition file for the target system is SAM286.DEF. Figure B-1 shows memory maps of the layout of both systems: the 28612.def layout is on the left and the target system layout is on the right.

B.4 STEPS PERFORMED TO CREATE THE TARGET SYSTEM

The steps needed to modify an existing definition file to meet the target system needs are outlined below.

- Add the iSBC 220 SMD Driver
- Add the iSBC 208 Flexible Disk Controller Driver
- Add the RAM Driver
- Change the memory for the system and Free Space Manager to reflect the amount of memory needed for operation of the new devices.

As you proceed through this example refer to the *Extended iRMX II Interactive Configuration Utility Reference* manual for more information about configuring each of the above drivers.

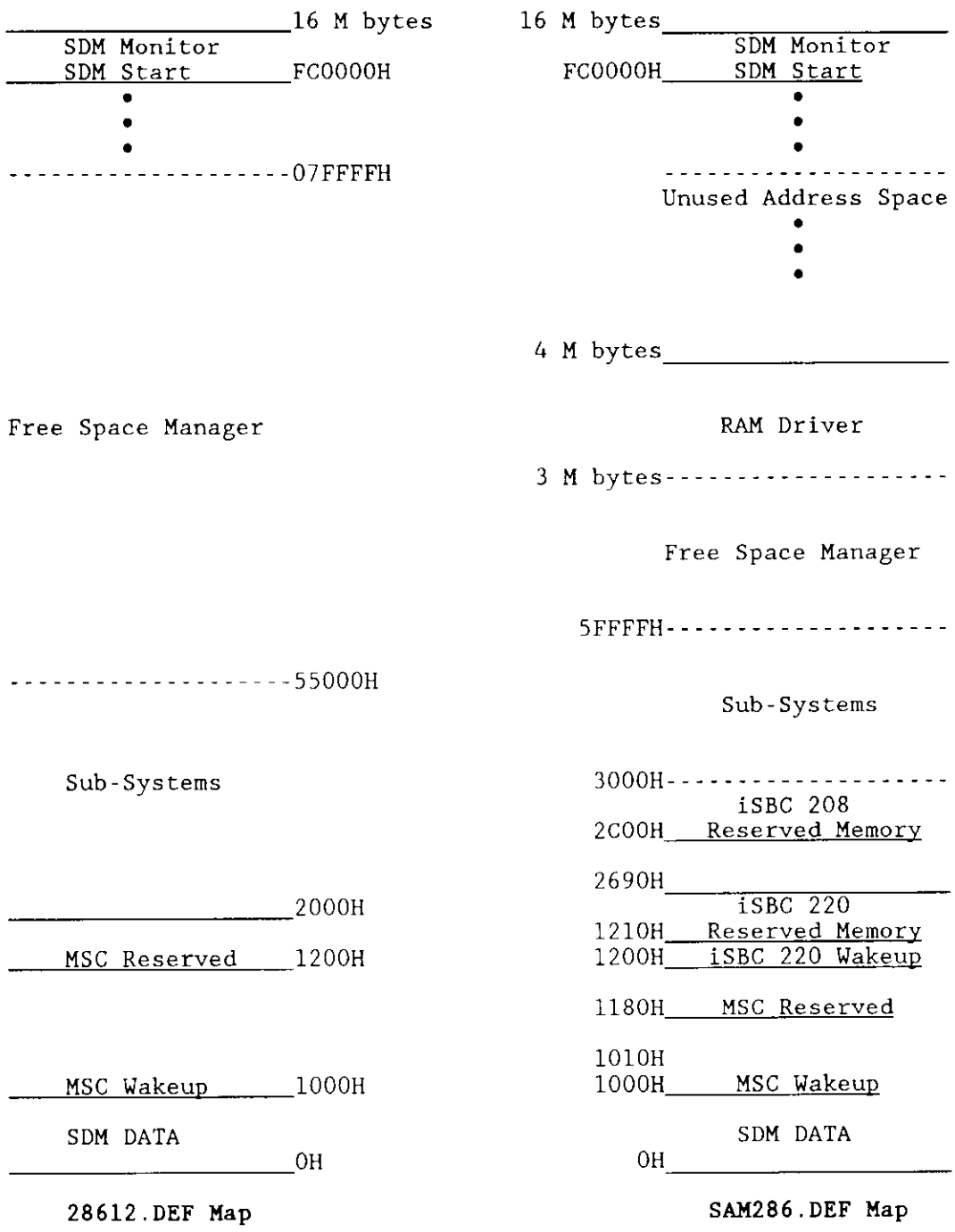


Figure B-1. Memory Maps for the 80286-Based System and Target System

B.5 USING THE ICU TO DEFINE THE TARGET SYSTEM

This section describes a dialogue between a user and the ICU. This dialogue demonstrates the steps needed to define the target system described in the previous section. In the dialogue, user input is shown in either blue or bolded text, followed by a carriage return (<CR>). Should you make an error in entering information as you proceed through this example, you can re-type the information if you are currently viewing the screen in which the error was entered. If not, you can use either the Backup (b) or Find (f) command to access the screen you want to change, then re-type the correct information. If you are new to the iRMX II ICU and do not want to use these commands, you can delete the SAM286.DEF file in the SAM286 directory and start over by entering the last line of the command sequence listed below.

Invoke the ICU, giving the name of the default file and the desired name of the modified definition file, as follows:

```
-
-
-
```

This produces the display shown in Figure B-2. This is the first screen you see each time you invoke the ICU.

```
iRMX II Interactive Configuration Utility For Extended iRMX II, <v>
Copyright <years> Intel Corporation


 For general help in any screen enter H <cr>.

The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

    ENTER COMMAND : C idevs <CR>
```

Figure B-2. Initial ICU Screen

The first thing to change in the target system is the MSC disk and tape controller. To do this, go to the "Intel Device Drivers" screen by entering "C idevs <CR>", as shown in Figure B-2. This produces the screen shown in Figure B-3.

```
(IDEVS)     Intel Device Drivers

(S14) Mass Storage Controller      [Yes/No]     YES
(T74) 8274 Terminal Driver         [Yes/No]     YES
(T51) 8251A Terminal Driver        [Yes/No]     NO
(T30) 82530 Terminal Driver        [Yes/No]     NO
(TCC) Terminal Comm Controller     [Yes/No]     YES
(L86) Line Printer for iSBC 286/10 [Yes/No]     YES
(L50) Line Printer - iSBX 350      [Yes/No]     NO
(S20) iSBC 220     [Yes/No]  NO   (X18) iSBX 218A        [Yes/No] NO
(S08) iSBC 208     [Yes/No]  NO   (T34) iSBC 534         [Yes/No] NO
(T44) iSBC 544A    [Yes/No]  YES  (S64) iSBC 264         [Yes/No] NO
(X51) iSBX 251     [Yes/No]  NO   (RAM) RAM Disk Driver  [Yes/No] NO
(SCS) SCSI Driver  [Yes/No]  NO   (S24) iSBC 186/224A    [Yes/No] NO
(S10) iSBC 186/410 [Yes/No]  NO   (G79) iSBX 279         [Yes/No] NO

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: S14 = y <CR>
```

Figure B-3. The Intel Device Drivers Screen

The "Intel Device Drivers" screen, shown in Figure B-3, lists all of the available Intel devices. The "YES" or "NO" field shown to the right of each device indicates whether or not it is part of the current definition file. To add a device or go to the first screen of an existing device, enter its three-letter abbreviation followed by "=y <CR>". The abbreviation for the S14 is "S14", so type "S14=Y <CR>". This produces the screen shown in Figure B-4.

The parameter that must be changed is the "IPA I/O Processor Block Address" parameter. The memory maps in Figure B-1 show why. From these maps you can see that there would be an address conflict between the iSBC 220 SMD controller and the original location of the MSC disk and tape controller. To prevent this conflict, one of the devices must be moved. Moving the "IPA I/O Processor Block Address" from 1200H to 1180H will resolve the conflict. Enter "ipa=1180H <CR>", as shown in Figure B-4. To check this change, type "<CR>". This produces the screen shown in Figure B-5.

```
(D214)              Mass Storage Controller Driver

(DEV)  Device Name [1-16 Chars]                        215-A
(IL)   Interrupt Level [Encoded Level]                 058H
(ITP)  Interrupt Task Priority [0-255]                 130
(WIP)  Wakeup I/O Port [0-0FFFFH]                      0100H
(IPA)  I/O Processor Block Address [0-0FFFFFH]         01200H

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: ipa=1180h <CR>
: <CR>
```

Figure B-4 MSC Driver Screen

```
(D214)              Mass Storage Controller Driver

(DEV)  Device Name [1-16 Chars]                        215-A
(IL)   Interrupt Level [Encoded Level]                 058H
(ITP)  Interrupt Task Priority [0-255]                 130
(WIP)  Wakeup I/O Port [0-0FFFFH]                      0100H
(IPA)  I/O Processor Block Address [0-0FFFFFH]         01180H

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: f idevs <CR>
```

Figure B-5 MSC Driver Screen

To add the iSBC 220 SMD controller, you return to the "Intel Device Drivers" screen by entering "f idevs <CR>". This produces the screen shown in Figure B-6.

```
(IDEVS)     Intel Device Drivers

(S14) Mass Storage Controller      [Yes/No]     YES
(T74) 8274 Terminal Driver         [Yes/No]     YES
(T51) 8251A Terminal Driver        [Yes/No]     NO
(T30) 82530 Terminal Driver        [Yes/No]     NO
(TCC) Terminal Comm Controller     [Yes/No]     YES
(L86) Line Printer for iSBC 286/10 [Yes/No]     YES
(L50) Line Printer - iSBX 350      [Yes/No]     NO
(S20) iSBC 220     [Yes/No]  NO   (X18) iSBX 218A       [Yes/No] NO
(S08) iSBC 208     [Yes/No]  NO   (T34) iSBC 534        [Yes/No] NO
(T44) iSBC 544A    [Yes/No]  YES  (S64) iSBC 264        [Yes/No] NO
(X51) iSBX 251     [Yes/No]  NO   (RAM) RAM Disk Driver [Yes/No] NO
(SCS) SCSI Driver  [Yes/No]  NO   (S24) iSBC 186/224A   [Yes/No] NO
(S10) iSBC 186/410 [Yes/No]  NO   (G79) iSBX 279        [Yes/No] NO

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: S20 = y <CR>
```

Figure B-6. The Intel Device Drivers Screen

The abbreviation for the iSBC 220 SMD controller is "S20", so you type "S20=y <CR>" at the bottom of the screen, as shown in Figure B-6. This will produce the screen shown in Figure B-7.

```
Do you want any/more iSBC 220 DEVICEs?
y <CR>
```

Figure B-7. Query Screen for the iSBC® 220 SMD Device

Figure B-7 shows a query screen that asks if you want to add a device. To start the process of adding the iSBC 220 SMD controller, type "y <CR>" as shown in Figure B-7. This will produce the "iSBC 220 Driver" screen, as shown in Figure B-8.

```
(D220)     iSBC 220 Driver

(DEV) Device Name [1-16 Chars]
(IL)  Interrupt Level [Encoded Level]          028H
(ITP) Interrupt Task Priority [0-0FFH]         130
(WIP) Wakeup I/O Port [0-0FFFFH]               0120H
(IPA) I/O Processor Block Address [0-0FFFFFH]  01210H
(SB)  Size of Buffers [0-0FFFFH]               01480H

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: dev=220 <CR>
: <CR>
```

Figure B-8. The iSBC® 220 Driver Screen

The minimum information that must be entered on this screen is the "(DEV) Device Name" field. Enter "dev=220 <CR>", as shown in Figure B-8. Because all of the remaining fields match the target system's hardware configuration for the iSBC 220 SMD controller, you can use the default values. Enter "<CR>" to reshow the screen with the name field completed. This produces the screen shown in Figure B-9.

```
(D220)     iSBC 220 Driver

(DEV) Device Name [1-16 Chars]                 220
(IL)  Interrupt Level [Encoded Level]          028H
(ITP) Interrupt Task Priority [0-0FFH]         130
(WIP) Wakeup I/O Port [0-0FFFFH]               0120H
(IPA) I/O Processor Block Address [0-0FFFFFH]  01210H
(SB)  Size of Buffers [0-0FFFFH]               01480H

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: <CR>
```

Figure B-9. The Completed iSBC® 220 Driver Screen

Check that you have entered the device name correctly. If so, you are ready to add the iSBC 220 unit information. Typing a "<CR>", as shown in Figure B-9, will display the query screen shown in Figure B-10.

```
Do you want any/more iSBC 220 DEVICEs ?
<CR>
```

Figure B-10. Query Screen for another iSBC® 220 Driver

As this application requires only one iSBC 220 SMD controller, respond with a carriage return (<CR>) as shown in Figure B-10. This tells the ICU that you do not want another iSBC 220 Driver and causes the next screen to be displayed, as shown in Figure B-11.

```
Do you want any/more iSBC 220 UNITs ?
y <CR>
```

Figure B-11. Query Screen for iSBC® 220 SMD Controller Unit Information

Figure B-11 shows a query screen that asks if you want to fill in Unit Information for an iSBC 220 SMD controller. This is the first time that an iSBC 220 SMD controller has been added, so such information does not yet exist. Respond to this screen by entering "y <CR>", as shown in Figure B-11. This produces the "iSBC 220 Unit Information" screen shown in Figure B-12.

```
(U220)     iSBC 220 Unit Information

(DEV) Device Name [1-16 Chars]
(NAM) Unit Info Name [1-16 Chars]
(MR)  Maximum Retries [0-0FFFFH]                    09H
(CS)  Cylinder Size [0-0FFFFH]                      07EH
(NC)  Number of Cylinders [0-0FFFFH]                024DH
(NFH) Number of Heads/Fixed Disk [0-0FFH]           07H
(NRH) Number of Heads/Removable Disk [0-0FFH]       0H
(NS)  Number of Sectors/Track [0-0FFFFH]            012H
(NAC) Number of Alternate Cylinders [0-0FFH]        0BH
(SSN) Starting Sector Number [0-0FFFFFFFFH]         0H
(BTI) Bad Track Information [Yes/No]                YES

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: dev=220 <CR>
: nam=uinfo_220 <CR>
: <CR>
```

Figure B-12. The iSBC® 220 Unit Information Screen

The fields requiring information on this screen are the "(DEV) Device Name" and "(NAM) Unit Info Name" fields. Enter "dev=220 <CR>" and "nam=uinfo_220 <CR>", as shown in Figure B-12. The remaining fields need not be changed because the defaults match the target system. Reshow the "iSBC 220 Unit Information" screen to ensure that you typed everything correctly by entering "<CR>", as shown in Figure B-12. This produces the screen shown in Figure B-13.

```
(U220)     iSBC 220 Unit Information

(DEV) Device Name [1-16 Chars]                    220
(NAM) Unit Info Name [1-16 Chars]                 UINFO_220
(MR)  Maximum Retries [0-0FFFFH]                  09H
(CS)  Cylinder Size [0-0FFFFH]                    07EH
(NC)  Number of Cylinders [0-0FFFFH]              024DH
(NFH) Number of Heads/Fixed Disk [0-0FFH]         07H
(NRH) Number of Heads/Removable Disk [0-0FFH]     0H
(NS)  Number of Sectors/Track [0-0FFFFH]          012H
(NAC) Number of Alternate Cylinders [0-0FFH]      0BH
(SSN) Starting Sector Number [0-0FFFFFFFFH]       0H
(BTI) Bad Track Information [Yes/No]              YES

Enter    [ Abbreviation = new/value / Abbreviation ? / H ]
: <CR>
```

Figure B-13. Completed iSBC® 220 Unit Information Screen

After checking the entries on the screen shown in Figure B-13, type a "<CR>" to view the next screen.

```
Do you want any/more iSBC 220 UNITs ?
<CR>
```

Figure B-14. iSBC® 220 Unit Query Screen

Only one Unit is required for the iSBC 220 SMD controller, so by entering a carriage return (<CR>), as shown in Figure B-14, you tell the ICU that you are ready to view the next major screen.

```
Do you want any/more iSBC 220 DUIBs ?
y <CR>
```

Figure B-15. iSBC® 220 SMD DUIB Query Screen

To complete the inclusion of the iSBC 220 SMD controller, a Device Unit Information Block (DUIB) must be completed. Typing "y <CR>", as shown in Figure B-15, causes the "iSBC 220 Device-Unit Information" screen in Figure B-16 to appear.

```
(I220)       iSBC 220 Device Unit Information

(DEV) Device Name [1-16 Chars]
(NAM) Device-Unit Name [1-14 Chars]
(PFD) Physical File Driver Required [Yes/No]                YES
(NFD) Named File Driver Required [Yes/No]                   YES
(GRA) Granularity [0-0FFFFH]                                0400H
(DSZ) Device Size [0-0FFFFFFFFH]                            0471F000H
(UN)  Unit Number on this Device [0-0FFH]                   0H
(UIN) Unit Info Name [1-16 Chars]
(RUT) Request Update Timeout [0-0FFFFH]                     064H
(NB)  Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]     08H
(CUP) Common Update [Yes/No]                                YES
(MB)  Max Buffers [0-0FFH]                                  0FFH

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: dev=220 <CR>
: nam=sf0 <CR>
: uin=uinfo_220 <CR>
: <CR>
```

Figure B-16. The iSBC® 220 Device-Unit Information (DUIB) Screen

Once again the default values match the target system. The only fields that must be filled in are the "(DEV) Device Name", "(NAM) Device-Unit Name", and "(UIN) Unit Info Name" fields. Enter "dev=220 <CR>", "nam=sf0 <CR>", and "uin=uinfo_220 <CR>", as shown in Figure B-16. To reshow the screen to check your entries, enter a carriage return (<CR>) as shown in Figure B-16. This produces the screen shown in Figure B-17.

```
(I220)       iSBC 220 Device Unit Information

(DEV) Device Name [1-16 Chars]                              220
(NAM) Device-Unit Name [1-14 Chars]                         SF0
(PFD) Physical File Driver Required [Yes/No]                YES
(NFD) Named File Driver Required [Yes/No]                   YES
(GRA) Granularity [0-0FFFFH]                                0400H
(DSZ) Device Size [0-0FFFFFFFFH]                            0471F000H
(UN)  Unit Number on this Device [0-0FFH]                   0H
(UIN) Unit Info Name [1-16 Chars]                           UINFO_220
(RUT) Request Update Timeout [0-0FFFFH]                     064H
(NB)  Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]     08H
(CUP) Common Update [Yes/No]                                YES
(MB)  Max Buffers [0-0FFH]                                  0FFH

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: f idevs  <CR>
```

Figure B-17. Completed iSBC® 220 Device-Unit Information Screen

If all the entries are correct, all of the steps to include the iSBC 220 SMD Controller are completed.

Note that adding the MSC disk and tape, the iSBC 220 SMD, and the iSBC 208 peripheral devices requires changing the amount of memory available to both the system, "(MEMS) Memory for System" screen, and the Free Space Manager, "(MEMF) Memory for Free Space Manager" screen. Each of these drivers requires RAM space in the low megabyte of memory for host/controller communications and, in the case of the iSBC 208 and iSBC 220, some I/O buffering. This memory must be taken from the Free Space Manager and given to the system. This process can be performed before or after adding the driver screens. For this example, all of the devices will be added, then the memory parameters will be changed to incorporate the new devices.

The next step, then, is to add the iSBC 208 Flexible Disk Drive Controller. To begin, return to the "Intel Device Drivers" screen by entering "f idevs <CR>" as shown in Figure B-17.

```
(IDEVS)     Intel Device Drivers

(S14) Mass Storage Controller       [Yes/No]     YES
(T74) 8274 Terminal Driver          [Yes/No]     YES
(T51) 8251A Terminal Driver         [Yes/No]     NO
(T30) 82530 Terminal Driver         [Yes/No]     NO
(TCC) Terminal Comm Controller      [Yes/No]     YES
(L86) Line Printer for iSBC 286/10 [Yes/No]     YES
(L50) Line Printer - iSBX 350       [Yes/No]     NO
(S20) iSBC 220      [Yes/No]  NO   (X18) iSBX 218A        [Yes/No] NO
(S08) iSBC 208      [Yes/No]  NO   (T34) iSBC 534          [Yes/No] NO
(T44) iSBC 544A     [Yes/No]  YES  (S64) iSBC 264          [Yes/No] NO
(X51) iSBX 251      [Yes/No]  NO   (RAM) RAM Disk Driver [Yes/No] NO
(SCS) SCSI Driver   [Yes/No]  NO   (S24) iSBC 186/224A    [Yes/No] NO
(S10) iSBC 186/410 [Yes/No]  NO   (G79) iSBX 279         [Yes/No] NO

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: S08 = y <CR>
```

Figure B-18. The Intel Device Drivers Screen

To add any driver from the (IDEVS) screen, you type the device's three-letter abbreviation and "=y <CR>". The abbreviation for the iSBC 208 Flexible Disk Drive Controller is "S08", so you type "S08 = y <CR>" as shown in Figure B-18. This produces the screen shown in Figure B-19.

```
Do you want any/more iSBC 208 DEVICEs?
y <CR>
```

Figure B-19. iSBC® 208 Device Query Screen

Figure B-19 shows a query screen that asks users if they want to add a device. To start the process of adding the iSBC 208 Flexible Disk Drive Controller, type "y <CR>". This causes the "iSBC 208 Driver" screen, as shown in Figure B-20, to appear.

```
(D208)     iSBC 208 Driver

(DEV) Device Name [1-16 Chars]
(IL)  Interrupt Level [Encoded Level]          048H
(ITP) Interrupt Task Priority [0-255]          130
(PA)  Port Address [0-0FFFFH]                  0180H
(MDV) Motor Delay Value [0-0FFFFH]             050H
(BBA) Boundary Buffer Address [0-0FFFFFFFFH]   01600H

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: dev=208 <CR>
: bba=2C00H <CR>
: <CR>
```

Figure B-20. The iSBC® 208 Driver Screen

Because the target system matches most of this screen's default values, only two fields need to be changed on this screen. The "(DEV) Device Name" and the "(BBA) Boundary Buffer Address" fields. Enter "dev=208 <CR>" to fill in the "(DEV)"field. Because the iSBC 220 driver has memory located at the default iSBC 208 address (see Figure B-42) the "(BBA)" field must be changed. Enter "bba=2C00H <CR>" to change this address. Enter a carriage return (<CR>) to reshow the screen (Figure B-21).

```
(D208)     iSBC 208 Driver

(DEV) Device Name [1-16 Chars]                 208
(IL)  Interrupt Level [Encoded Level]          048H
(ITP) Interrupt Task Priority [0-255]          130
(PA)  Port Address [0-0FFFFH]                  0180H
(MDV) Motor Delay Value [0-0FFFFH]             050H
(BBA) Boundary Buffer Address [0-0FFFFFFFFH]   2C00H

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: <CR>
```

Figure B-21. Completed iSBC® 208 Driver Screen

The changes and additions needed on this screen have been made and checked; the "iSBC 208 Unit Information" screen must now be filled in. To indicate to the ICU that you are ready to view the next screen, enter a carriage return (<CR>) as shown in Figure B-21.

```
Do you want any/more iSBC 208 DEVICEs?
<CR>
```

Figure B-22. Query Screen for another iSBC® 208 Driver

As this application requires only one iSBC 208 flexible disk driver, respond with a carriage return as shown in Figure B-22. This tells the ICU that you do not want another iSBC 208 driver and causes the iSBC 208 Unit query screen to appear, as shown in Figure B-23.

```
Do you want any/more iSBC 208 UNITs?
y <CR>
```

Figure B-23. Query Screen for the iSBC® 208 Unit Information

Because this is a newly added device, you must complete all screens related to the iSBC 208 Flexible Disk Drive Controller. Respond to this screen by entering "y <CR>", as shown in Figure B-23 to produce the screen shown in Figure B-24.

```
(U208) iSBC 208 Unit Information

(DEV) Device Name [1-16 Chars]
(NAM) Unit Info Name [1-16 Chars]
(MR)  Maximum Retries [0-0FFFFH]              09H
(CS)  Cylinder Size [0-0FFFFH]                01AH
(NT)  Number of Tracks per Side [0-0FFFFH]    04DH
(NS)  Number of Sectors/Track [0-0FFFFH]      01AH
(SR)  Step Rate [0-0FFH]                      08H
(HLT) Head Load Time [0-0FFH]                 028H
(HUT) Head Unload Time [0-0FFH]               0F0H

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: dev=208 <CR>
: nam=uinfo_208 <CR>
: <CR>
```

Figure B-24. iSBC® 208 Unit Information Screen

Because the target system matches the defaults on this screen the only fields you must enter are the "(DEV) Device Name" and "(NAM) Unit Info Name" fields. Enter "dev=208 <CR>" and "nam=uinfo_208 <CR>", as shown in Figure B-24. To reshow the screen to check your entries, enter a carriage return (<CR>).

```
(U208) iSBC 208 Unit Information

(DEV)  Device Name [1-16 Chars]                  208
(NAM) Unit Info Name [1-16 Chars]                UINFO_208
(MR)  Maximum Retries [0-0FFFFH]                 09H
(CS)  Cylinder Size [0-0FFFFH]                   01AH
(NT)  Number of Tracks per Side [0-0FFFFH]       04DH
(NS)  Number of Sectors/Track [0-0FFFFH]         01AH
(SR)  Step Rate [0-0FFH]                         08H
(HLT) Head Load Time [0-0FFH]                    028H
(HUT) Head Unload Time [0-0FFH]                  0F0H

Enter   [ Abbreviation = new_value / Abbreviation ? / H ]
: <CR>
```

®Figure B-25. Completed iSBC® 208 Unit Information Screen

Figure B-25 shows the completed "iSBC 208 Unit Information" screen. After checking that the entries are correct, enter a carriage return (<CR>) to continue completing the screens related to the iSBC 208 Flexible Disk Drive Controller.

```
Do you want any/more iSBC 208 UNITs ?
<CR>
```

Figure B-26. iSBC® 208 Unit Query Screen

This application requires only one Unit. To continue completing the screens related to the iSBC 208 Flexible Disk Drive Controller, enter a carriage return (<CR>), as shown in Figure B-26.

```
Do you want any/more iSBC 208 DUIBs ?
y <CR>
```

Figure B-27. The iSBC® 208 DUIB Query Screen

To complete the addition of the iSBC 208 Flexible Disk Drive Controller, you must complete a Device Unit Information Block (DUIB). Typing a "y <CR>" causes the "iSBC 208 Device-Unit Information" screen, shown in Figure B-28, to appear.

```
(I208)  iSBC 208 Device-Unit Information

(DEV) Device Name [1-16 Chars]
(NAM) Device-Unit Name [1-14 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]        DOUBLE
(SDS) Single or Double Sided Disks [Single/Double]          SINGLE
(EFI) 8 or 5 Inch Disks [8/5]                               8
(SUF) Standard or Uniform Format [Standard/Uniform]         STANDARD
(GRA) Granularity [0-0FFFFH]                                0100H
(DSZ) Device Size [0-0FFFFFFFFH]                            07C500H
(UN)  Unit Number on this Device [0-0FFH]                   0H
(UIN) Unit Info Name [1-16 Chars]
(RUT) Request Update Timeout [0-0FFFFH]                     064H
(NB)  Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]     06H
(CUP) Common Update [Yes/No]                                YES
(MB)  Max Buffers [0-0FFH]                                  0FFH

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: dev=208 <CR>
: nam=afd0 <CR>
: uin=uinfo_208 <CR>
: <CR>
```

Figure B-28. The iSBC® 208 Device-Unit Information (DUIB) Screen

The default values provided on this screen match the target system for this application. The only fields that must be filled in are "(DEV) Device Name", "(NAM) Device-Unit Name", and "(UIN) Unit Info Name". Enter "dev=208", "nam=afd0", and "uin=uinfo_208", as shown in Figure B-28. Reshow the screen to check your entries by entering a carriage return (<CR>).

```
(I208)  iSBC 208 Device-Unit Information

(DEV) Device Name [1-16 Chars]                                  208
(NAM) Device-Unit Name [1-14 Chars]                             AFD0
(PFD) Physical File Driver Required [Yes/No]                    YES
(NFD) Named File Driver Required [Yes/No]                       YES
(SDD) Single or Double Density Disks [Single/Double]            DOUBLE
(SDS) Single or Double Sided Disks [Single/Double]              SINGLE
(EFI) 8 or 5 Inch Disks [8/5]                                   8
(SUF) Standard or Uniform Format [Standard/Uniform]             STANDARD
(GRA) Granularity [0-0FFFFH]                                    0100H
(DSZ) Device Size [0-0FFFFFFFFH]                                07C500H
(UN)  Unit Number on this Device [0-0FFH]                       0H
(UIN) Unit Info Name [1-16 Chars]                               UINFO_208
(RUT) Request Update Timeout [0-0FFFFH]                         064H
(NB)  Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]         06H
(CUP) Common Update [Yes/No]                                    YES
(MB)  Max Buffers [0-0FFH]                                      0FFH

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: f idevs <CR>
```

Figure B-29. Completed iSBC® 208 Device-Unit Information Screen

All of the steps to include the iSBC 208 flexible disk driver are completed. You still, however, must adjust the system memory to allow for the addition of this device. For now, though, add the RAM driver to complete the addition of all the devices listed in the changes. The first step is to return to the "Intel Device Drivers" screen by entering "f idevs <CR>" as shown in Figure B-29.

```
(IDEVS)     Intel Device Drivers

(S14) Mass Storage Controller      [Yes/No]     YES
(T74) 8274 Terminal Driver         [Yes/No]     YES
(T51) 8251A Terminal Driver        [Yes/No]     NO
(T30) 82530 Terminal Driver        [Yes/No]     NO
(TCC) Terminal Comm Controller     [Yes/No]     YES
(L86) Line Printer for iSBC 286/10 [Yes/No]     YES
(L50) Line Printer - iSBX 350      [Yes/No]     NO
(S20) iSBC 220     [Yes/No]  NO   (X18) iSBX 218A       [Yes/No] NO
(S08) iSBC 208     [Yes/No]  NO   (T34) iSBC 534        [Yes/No] NO
(T44) iSBC 544A    [Yes/No]  YES  (S64) iSBC 264        [Yes/No] NO
(X51) iSBX 251     [Yes/No]  NO   (RAM) RAM Disk Driver [Yes/No] NO
(SCS) SCSI Driver  [Yes/No]  NO   (S24) iSBC 186/224A   [Yes/No] NO
(S10) iSBC 186/410 [Yes/No]  NO   (G79) iSBX 279        [Yes/No] NO

Enter   [ Abbreviation = new/value / Abbreviation ? / H ]
: ram=y <CR>
```

Figure B-30. Intel Device Drivers Screen

To add the RAM driver, you type the device's three-letter abbreviation from the (IDEVS) screen and "=y <CR>". The abbreviation for the RAM Disk is "RAM", so you type "ram= y <CR>" as shown in Figure B-30. This produces the screen in Figure B-31.

```
Do you want any/more RAM Disk Driver DEVICEs?
y <CR>
```

Figure B-31. RAM Disk Device Query Screen

To start adding the RAM disk, type "y <CR>". This entry produces the "RAM Disk Driver" screen as shown in Figure B-32.

```
(DRAM)      RAM Disk Driver

(DEV) Device Name [1-16 Characters]

Enter [Abbreviation = new_value / Abbreviation ? / H ]:
: dev=ram <CR>
: <CR>
```

Figure B-32. Default RAM Disk Driver Screen

The only information to enter on this screen is the "(DEV) Device Name" field. Enter "dev= ram <CR>", as shown in Figure B-32. Enter a carriage return (<CR>) to redisplay the screen with the new data.

```
(DRAM)      RAM Disk Driver

(DEV) Device Name [1-16 Characters]      RAM

Enter [Abbreviation = new_value / Abbreviation ? / H ]:
: <CR>
```

Figure B-33. Inserted RAM Disk Driver Screen

Figure B-33 shows that the necessary change has been made; therefore, simply enter a carriage return (<CR>) to view the next query screen as shown in Figure B-34.

```
Do you want any/more RAM Disk Driver DEVICEs ?
<CR>
```

Figure B-34. RAM Driver Query Screen

Because you have only one RAM disk driver to add, enter a carriage return (<CR>) to view the RAM driver Units query screen as shown in Figure B-35.

```
Do you want any/more RAM Disk Driver UNITs ?
y <CR>
```

Figure B-35. RAM Unit Query Screen

Because the Unit information does not yet exist in our definition file, type "y <CR>". This produces the "RAM Disk Driver Unit Information" screen as shown in Figure B-36.

```
(URAM)   RAM Disk Driver Unit Information

(DEV) Device Name [1-16 Characters]
(NAM) Unit Info Name [1-16 Chars]
(BMA) Base Memory Address [0-0FFFFFFH]          0100000H
(WP)  Write Protected [Yes/No]                  NO

Enter [Abbreviation = new_value / Abbreviation ? / H ] :
: dev=ram <CR>
: nam=uinfo_ram CR>
: bma=300000h  <CR>
: <CR>
```

Figure B-36. RAM Disk Driver Unit Information Screen

The fields that must be filled in on this screen are "(DEV) Device Name", "(NAM) Unit Info Name", and "(BMA) Base Memory Address". The base memory address value is chosen by selecting an address above the default memory reserved for the system and the Free Space Manager in the memory map shown in Figure B-42. Enter "dev=ram", "nam=uinfo_ram", and "bma=300000h". To check your entries, enter a "<CR>". This action produces the screen shown in Figure B-37.

```
(URAM)   RAM Disk Driver Unit Information

(DEV) Device Name [1-16 Characters]                          RAM
(NAM) Unit Info Name [1-16 Chars]                            UINFO_RAM
(BMA) Base Memory Address [0-0FFFFFFH]                       300000H
(WP)  Write Protected [Yes/No]                               NO

Enter [Abbreviation = new_value / Abbreviation ? / H ] :
: <CR>
```

Figure B-37. Inserted RAM Disk Driver Unit Information Screen

After checking the entries on the screen shown in Figure B-37, type a "<CR>" to view the next screen.

```
Do you want any/more RAM Disk Driver UNITs ?
<CR>
```

Figure B-38. RAM Unit Query Screen

Only one Unit is required for the RAM Disk, so by entering a "<CR>", as shown in Figure B-38, you continue the process of adding the RAM Disk.

```
Do you want any/more RAM Disk Driver DUIBs ?
y <CR>
```

Figure B-39. RAM DUIB Query Screen

To finish adding the RAM disk, you must complete a DUIB. To begin, enter "y <CR>" as shown in Figure B-39. The ICU then displays the "Ram Disk Device-Unit Information" screen, as shown in Figure B-40.

```
(IRAM)   RAM Disk Device-Unit Information

(DEV) Device Name [1-16 Characters]
(NAM) Device-Unit Name [1-14 chars]
(PFD) Physical File Driver Required [Yes/No]                YES
(NFD) Name File Driver Required [Yes/No]                    YES
(GRA) Granularity [0-0FFFFH]                                0200H
(DSZ) Device Size [0-0FFFFFFFFH]                            0100000H
(UN)  Unit Number of this Device [0-0FFH]                   0H
(UIN) Unit Info Name [1-16 Chars]
(RUT) Request Update Timeout [0-0FFFFH]                     0H
(NB)  Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]     02H
(CUP) Common Update [Yes/No]                                YES
(MB)  Max Buffers [0-0FFH]                                  0FFH

Enter [Abbreviation = new_value / Abbreviation ? / H ] :
: dev=ram <CR>
: nam=r0  <CR>
: uin=uinfo_ram <CR>
: <CR>
```

Figure B-40. RAM Disk Device-Unit Information Screen

The fields that must be filled in on this screen are "(DEV) Device Name", "(NAM) Device-Unit Name", and "(UIN) Unit Info Name". The remaining fields can use the default values. Enter "dev=ram <CR>", "nam=r0", and "uin=uinfo_ram <CR>". To check your entries, type "<CR>". Figure B-41 shows the inserted "RAM Disk Device-Unit Information" screen.

```
(IRAM)   RAM Disk Device-Unit Information

(DEV) Device Name [1-16 Characters]                          RAM
(NAM) Device-Unit Name [1-14 chars]                          R0
(PFD) Physical File Driver Required [Yes/No]                 YES
(NFD) Name File Driver Required [Yes/No]                     YES
(GRA) Granularity [0-0FFFFH]                                 0200H
(DSZ) Device Size [0-0FFFFFFFFH]                             0100000H
(UN)  Unit Number of this Device [0-0FFH]                    0H
(UIN) Unit Info Name [1-16 Chars]                            UINFO_RAM
(RUT) Request Update Timeout [0-0FFFFH]                      0H
(NB)  Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]      02H
(CUP) Common Update [Yes/No]                                 YES
(MB)  Max Buffers [0-0FFH]                                   0FFH

Enter [Abbreviation = new_value / Abbreviation ? / H ] :
: f mems <CR>
```

Figure B-41. Inserted RAM Disk DUIB Information Screen

As mentioned earlier, adding certain devices requires adjustments to the memory for the system and the Free Space Manager. Use the memory map in Figure B-42 to determine where you will increase the memory locations for the system and decrease the memory locations for the Free Space Manager. Figures B-43 through B-48 show these changes. You will notice that you cannot simply insert the memory locations needed. First you must delete the current reserved memory locations, view the screen again to enter the new values, then view the screen a third time to check the changes. This procedure is necessary because the ICU does not accept addresses that could cause overlapping memory locations.

Enter "f mems <CR>" to view the "Memory for System" screen shown in Figure B-43.

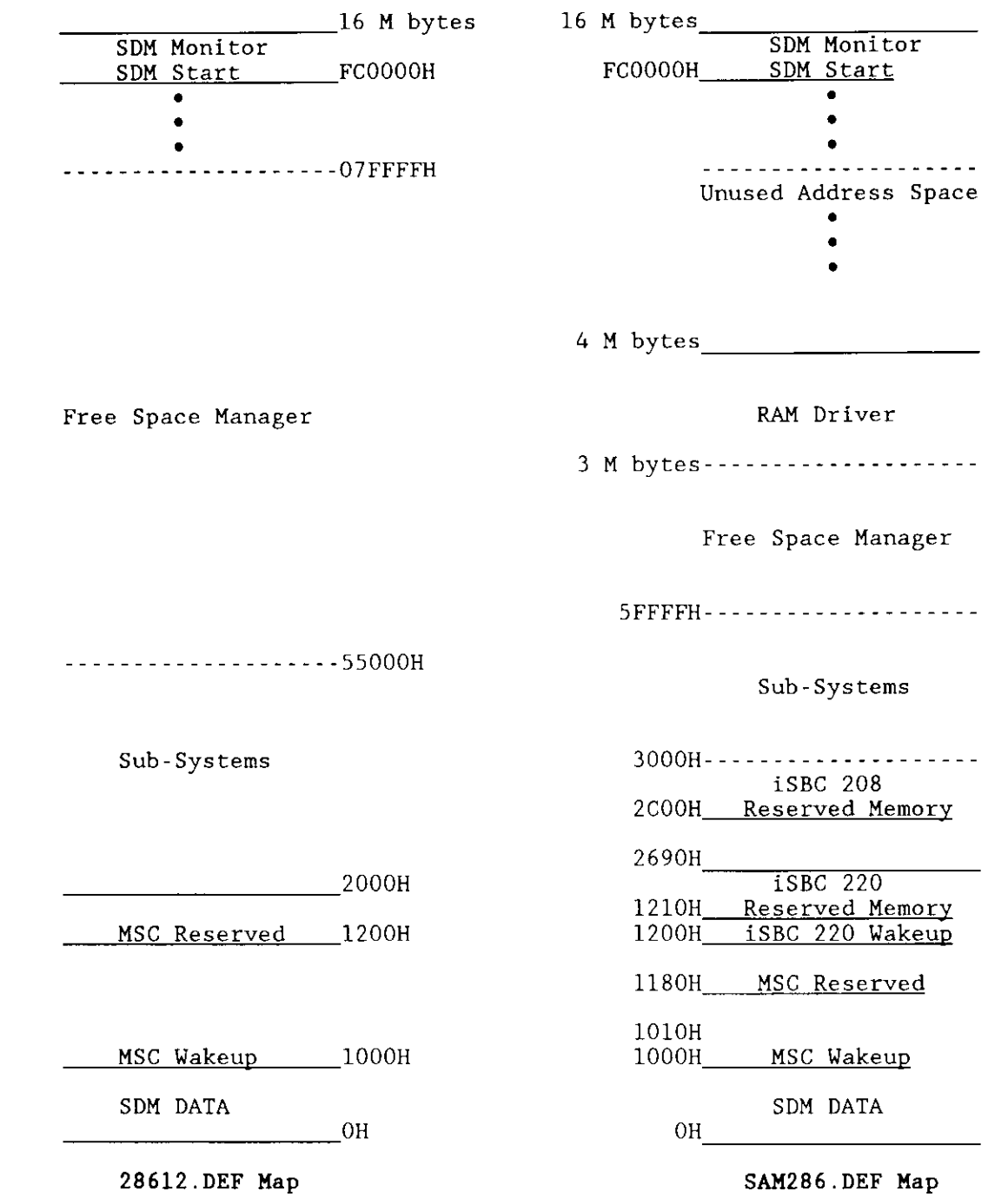


Figure B-42. The Original and Modified Memory Maps

```
(MEMS)          Memory for System

    SYS     = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =           02000H,          059FFFH
[2] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
: ^d 1 <CR>
```

Figure B-43. Memory for System Screen

To delete the default values type "^d 1 <CR>". This entry produces the screen shown in Figure B-44.

```
(MEMS)          Memory for System

    SYS     = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =

Enter Changes [Number = new_value /  ^D Number / ? / H ] :
: 1 = 3000h,5ffffh  <CR>
```

Figure B-44. Changing the Memory for System Screen

From the memory map in Figure B-42, you find the lower and upper address limits for system memory and configure them by entering "1=3000h,5ffffh <CR>", as shown in Figure B-44. This produces the "Memory for System" screen containing these changes, shown in Figure B-45. Note that the upper value (5FFFFH) was selected to ensure adequate memory space. If saving memory is a concern, examine the .mp2 file produced during system generation (G[enerate] command) to find out the actual system size. Once the actual system size is determined, adjust this address accordingly.

```
(MEMS)          Memory for System

    SYS     = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =           03000H,          05FFFFH
[2] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
: f memf <CR>
```

Figure B-45. New Memory for System Screen

Now, you must change the memory for the Free Space Manager. Enter "f memf <CR>" to view the "Memory for Free Space Manager" screen as shown in Figure B-46. Follow the steps for changing memory addresses, as shown in Figures B-46 through B-48.

```
(MEMF)          Memory for Free Space Manager

    FSM     = low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM     =          05A000H,         07FFFFFH
[2] FSM     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
: ^d 1 <CR>
```

Figure B-46. Memory for Free Space Manager Screen

```
(MEMF)          Memory for Free Space Manager

    FSM     = low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
: 1 = 60000h,02fffffh  <CR>
```

Figure B-47. Changing the Memory for Free Space Manager Screen

```
(MEMF)        Memory for Free Space Manager

    FSM    - low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM    -          060000H,        02FFFFFH
[2] FSM    -

Enter Changes [Number = new_value / ^D Number / ? / H ] :
: f gen <CR>
```

Figure B-48. New Memory for Free Space Manager Screen

Now that the system and Free Space Manager memory for the 80286-based multi-user system have been changed to match the target system, you are ready to generate the system. To produce the "Generate File Names" screen, type "f gen <CR>" as shown in Figure B-48.

```
(GEN)          Generate File Names

File Name [1-55 Characters]

(ROF) ROM Code File Name
                                              /BOOT/RMX286.ROM
(RAF) RAM Code File Name
                                              /BOOT/28612.286

Enter  [Abbreviation = new_value / Abbreviation ? / H ] :
: raf=/boot/sam286.286 <CR>
: <CR>
```

Figure B-49. Generate File Names Screen

On the "Generate File Names" screen, shown in Figure B-49, you must specify the pathname of the new bootloadable file you will be generating. For this application, the file is SAM286.286, and it will be a RAM code file name. Enter "raf=/boot/sam286.286 <CR>". To produce the changed "Generate File Names" screen (shown in Figure B-50), enter a carriage return "<CR>".

You are now ready to start the generation phase of the ICU. To do this you must first return to the ICU menu screen. Enter "c <CR>" on the screen shown in Figure B-50. This entry produces the ICU menu screen shown in Figure B-51.

```
(GEN)          Generate File Names

File Name [1-55 Characters]

(ROF) ROM Code File Name
                                                    /BOOT/RMX286.ROM
(RAF) RAM Code File Name
                                                    /BOOT/SAM286.286
Enter  [Abbreviation = new_value / Abbreviation ? / H ] :
: c <CR>
```

Figure B-50. New Generate File Names Screen

```
 For general help in any screen enter H <CR>.

The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND : g <CR>
```

Figure B-51. ICU Menu Screen

Entering the G command on the ICU menu screen produces the screen shown in Figure B-51, starting the generation process. As shown in Figure B-52, the display prompts you for the prefix you wish to assign to all the files generated by this submit file. For this example, type "s <CR>". The ICU echoes the prefix and informs you of the progress of the generation process.

```
ENTER a letter to be used as prefix: s <CR>

The prefix letter is: S
Beginning NUCLEUS File Generation
.....DONE
Beginning BIOS File Generation
.....DONE
Beginning EIOS File Generation
.....DONE
Beginning LOADER File Generation
.....DONE
Beginning HI File Generation
.....DONE
Beginning UDI File Generation
.....DONE
Beginning SDB File Generation
.....DONE
Beginning Submit File Generation
.....DONE
Beginning Build File Generation
.....DONE
NOTE:  To Generate your system submit    SAM286.CSD

 For general help in any screen enter H <CR>.


The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND : e <CR>
```

Figure B-52. Generation Phase ICU Screen

When the generation process is completed, the ICU displays the name of the resulting submit file. In this case, the submit file is SAM286.CSD. The ICU then continues automatically to the ICU menu screen where you should enter "e <CR>" to exit the ICU and save the definition file. Upon receiving the Exit command, the ICU informs you that the definition file has been written. It issues the following message before returning control to the command line:

```
The Definition File has been written to file:  SAM286.DEF
```

You are now ready to invoke the SUBMIT file SAM286.CSD. You do this by entering

-

The SUBMIT file assembles all the configuration files generated by the ICU and binds the object files with all the libraries required by the subsystems. It then builds the system. Figure B-53 shows a listing of the output from the SUBMIT file. You may notice warning messages. The warning messages are normal and can be ignored. Only error messages must be heeded.

```
- ;
- ;    NUCLEUS
- ;
- :LANG:ASM286 SNTABL.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:ASM286 SNUCDA.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:ASM286 SNJOBC.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:BND286 &
        .
        .
        .
**OBJECT(SNUC1.LNK) NODEBUG NOTYPE SEGSIZE(STACK(200)) &
** NOLOAD  NOPUBLICS EXCEPT (RQ_nil_os_ext  , &
        .
        .
        .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      2 WARNINGS,      0 ERRORS
- :LANG:BND286 &
        .
        .
        .
**OBJECT(SNUCLS.LNK) NODEBUG NOTYPE &
** NOLOAD SEGSIZE(STACK(200))
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      1 WARNING,      0 ERRORS
```

Figure B-53. Output of Submit File for SAM286.CSD

```
- ;
- ;     BIOS
- ;
- :LANG:ASM286 SITABL.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:ASM286 SICDEV.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:ASM286 SITDEV.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:bnd286 &
        .
        .
        .
**OBJECT (SIOS1.LNK) NODEBUG NOTYPE     &
** NOLOAD NOPUBLICS EXCEPT (rqaiosinittask , &
        .
        .
        .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      1 WARNING,      0 ERRORS
- :LANG:bnd286 &
        .
        .
        .
**OBJECT (STSC.LNK)  NODEBUG NOTYPE SEGSIZE(STACK(0))   &
** NOLOAD NOPUBLICS EXCEPT( TSCINITIO, &
        .
        .
        .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      0 WARNINGS,      0 ERRORS
```

Figure B-53. Output of Submit File for SAM286.CSD (continued)

```
- :LANG:bnd286 &
        .
        .
        .
**OBJECT (SIOS.LNK)  NODEBUG NOTYPE SEGSIZE(STACK(0))   &
** NOLOAD NOPUBLICS EXCEPT (rqaiosinittask , &
        .
        .
        .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      0 WARNINGS,      0 ERRORS
- ;
- ;    EIOS
- ;
- :LANG:ASM286 SETABL.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:ASM286 SEDEVC.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:ASM286 SEJOBC.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:BND286 &
        .
        .
        .
**OBJECT( SEIOS1.LNK) NOLOAD NODEBUG SEGSIZE(STACK(0)) &
** NOPUBLICS EXCEPT(rqeiosinittask , &
        .
        .
        .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      1 WARNING,      0 ERRORS
```

Figure B-53. Output of Submit File for SAM286.CSD (continued)

```
- :LANG:BND286 &
          .
          .
          .
**OBJECT( SEIOS.LNK) NOLOAD NODEBUG SEGSIZE(STACK(0))
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.       0 WARNINGS,       0 ERRORS
- ;
- ;     LOADER
- ;
- :LANG:ASM286 SLTABL.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,    NO WARNINGS,    NO ERRORS
- :LANG:PLM286 SLCONF.P28

iRMX II PL/M-286 COMPILER V2.5
Copyright Intel Corporation 1982, 1983, 1984, 1985
PL/M-286 COMPILATION COMPLETE.       0 WARNINGS,      0 ERRORS

- :LANG:BND286 &
          .
          .
          .
**OBJECT (SLOADR.LNK)  NOLOAD NODEBUG SEGSIZE(STACK(0))  &
** NOPUBLICS EXCEPT (asynchload,       &
          .
          .
          .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.       0 WARNINGS,       0 ERRORS
- ;
- ;     HI
- ;
- :LANG:PLM286 SHCONF.P28

iRMX II PL/M-286 COMPILER V2.5
Copyright Intel Corporation 1982, 1983, 1984, 1985
PL/M-286 COMPILATION COMPLETE.       1 WARNING,       0 ERRORS
```

Figure B-53. Output of Submit File for SAM286.CSD (continued)

```
- :LANG:BND286   &
          .
          .
          .
**OBJECT( SHI.LNK) NOLOAD NODEBUG SEGSIZE(STACK(0)) &
** NOPUBLICS EXCEPT(rqhiinittask,          &
          .
          .
          .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.       1 WARNING,       0 ERRORS
- ;
- ;    CLI
- ;
- :LANG:BND286           &
          .
          .
          .
**OBJECT(SCLI.LNK) NOLOAD NODEBUG            &
**SEGSIZE(STACK(02400H)) NOPUBLICS EXCEPT(hcliinit)
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.       0 WARNINGS,      0 ERRORS
- ;
- ;    UDI
- ;
- :LANG:ASM286 SUTABL.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:BND286 &
          .
          .
          .
**OBJECT( SUDI.LNK ) NODEBUG NOTYPE SEGSIZE(STACK(0)) &
**NOLOAD NOPUBLICS EXCEPT(U_Allocate ,                &
          .
          .
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.       0 WARNINGS,      0 ERRORS
```

Figure B-53. Output of Submit File for SAM286.CSD (continued)

```
- ;
- ;    SDB
- ;
- :LANG:ASM286 SSDBCN.A28
iRMX II iAPX286 MACRO ASSEMBLER, V1.3
Copyright 1982 Intel Corporation

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
- :LANG:BND286 &
          .
          .
          .
**OBJECT( SSDB.LNK) SS(STACK(0)) NOLOAD     &
**NOPUBLICS EXCEPT(rqsdbinittask )
iRMX II iAPX286 BINDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      0 WARNINGS,      0 ERRORS
- ;
- ;    BUILD
- ;
- :LANG:BLD286 &
          .
          .
          .
**    OBJECT (/RMX286/ICU/SAM286/SAM286.286 ) NODEBUG NOTYPE  &
**    BUILDFILE(SAM286.BLD)
iRMX II iAPX286 SYSTEM BUILDER, V3.2
Copyright 1982, 1985 Intel Corporation

PROCESSING COMPLETED.      1 WARNING,      0 ERRORS
- END SUBMIT sam286.CSD
```

Figure B-53. Output of Submit File for SAM286.CSD (continued)

This ends the output from the SUBMIT file SAM286.CSD. To bootstrap load this system, prepare a third-stage bootstrap loader by entering

This places a copy of the device-specific third-stage bootstrap loader in a boot file named /BOOT/SAM286, which is required to bootstrap load the bootable file for this example configuration.

If conserving memory is a consideration, you can minimize the memory assigned to the system by examining the segment map of file /BOOT/SAM286.MP2 created during the build process. By adding the segment limits in the segment map you can calculate the precise memory addresses required by the system and the Free Space Manager. Remember that when you configured the ICU, you estimated the memory required, leaving extra room for any changes made during development. Now with the exact addresses available you can minimize the memory you have reserved. For the rest of this example assume that this calculation resulted in a system that can reside in addresses 3000H to 59FFFH. To minimize the memory, you must invoke the ICU with the definition file you have just created. Invoke the ICU by entering

```
iRMX II Interactive Configuration Utility For Extended iRMX II, <v>
Copyright <years> Intel Corporation

 For general help in any screen enter H <CR>.

The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND : c mems <CR>
```

Figure B-54. Initial ICU Screen

This entry produces the initial ICU screen shown in Figure B-54. Entering "c mems <CR>" takes you to the "Memory for System" screen, shown in Figure B-55, where you can adjust the memory reserved for the system. Using the MP2 file you can determine that the system requires less memory than you had previously reserved, leaving more memory for the Free Space Manager. Consequently, you must change the memory screens, as shown in Figures B-55 through 60. Remember that the ICU does not accept overlapping memory addresses. Therefore, you must perform the separate steps of deleting the existing memory locations, making your changes, and viewing the new screen.

```
(MEMS)          Memory for System

    SYS     = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =          03000H,           05FFFFH
[2] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:^d 1 <CR>
```

Figure B-55. Memory for System Screen

```
(MEMS)          Memory for System

    SYS     = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:1= 3000h,5Bfffh <CR>
```

Figure B-56. Adjusting the Memory for System Screen

```
(MEMS)          Memory for System

    SYS     = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =          03000H,           059FFFH
[2] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:f memf <CR>
```

Figure B-57. Final Memory for System Screen

Once you have changed the "Memory for System" screen, you are ready to change the "Memory for Free Space Manager" screen. Once you know the system memory requirements, the Free Space Manager gets the rest of the memory. You can assign the Free Space Manager more memory than you had originally assigned since your system was smaller than you expected. Change the "Memory for Free Space Manager" screen, as shown in Figures B-58 through B-60.

```
(MEMF)          Memory For Free Space Manager

    FSM     = low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM     =          060000H,        02FFFFFH
[2] FSM     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:^d 1 <CR>
```

Figure B-58. Memory for Free Space Manager Screen

```
(MEMF)          Memory For Free Space Manager

    FSM     = low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:1 = 5C000h, 2fffffh  <CR>
```

Figure B-59. Changes to the Memory for Free Space Manager Screen

```
(MEMF)          Memory For Free Space Manager

    FSM     = low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM     =          05A000H,        02FFFFFH
[2] FSM     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:c <CR>
```

Figure B-60. Final Memory for Free Space Manager Screen

After you have made all the necessary changes to your definition file, you can save the file and exit the ICU. However, first you must return to the ICU menu screen (shown in Figure B-61) by entering "c <CR>" as the final entry in Figure B-60. As an extra precaution before exiting, you should list the contents of our definition file to a file or to a device. The List command lists the contents of the screens in the system you have just generated and enables you to ensure that you have the correct parameter values before saving the files. To use the List command, enter "l <file name> <CR>" in Figure B-61, which produces the screen displayed in Figure B-62.

```
    For general help in any screen enter H <CR>.

   The following commands are available

   Change
   Generate
   List
   Save
   Quit
   Exit
   Replace
   Detail-Level
   Backup

    ENTER COMMAND :1 sam286.lst <CR>
```

Figure B-61. Entering the List Command

```
The Definition File has been listed to file:     SAM286.LST

    For general help in any screen enter H <CR>.

   The following commands are available

   Change
   Generate
   List
   Save
   Quit
   Exit
   Replace
   Detail-Level
   Backup

    ENTER COMMAND : g <CR>
```

Figure B-62. ICU Menu Screen

Enter the G command on the ICU menu screen to start the generation process, shown in Figure B-62. As shown in Figure B-63, the display prompts you for the prefix you wish to assign to all the files generated by this submit file. For this example, type "s <CR>". The ICU echoes the prefix and informs you of the progress of the generation process.

```
ENTER a letter to be used as prefix: s <CR>

The prefix letter is: S
Beginning NUCLEUS File Generation
.....DONE
Beginning BIOS File Generation
.....DONE
Beginning EIOS File Generation
.....DONE
Beginning LOADER File Generation
.....DONE
Beginning HI File Generation
.....DONE
Beginning UDI File Generation
.....DONE
Beginning SDB File Generation
.....DONE
Beginning Submit File Generation
.....DONE
Beginning Build File Generation
.....DONE
NOTE:  To Generate your system submit    SAM286.CSD

 For general help in any screen enter H <CR>.


The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND : e <CR>
```

Figure B-63. Generation Phase ICU Screen

Now that you have L[isted] the definition file and G[enerated] the system configuration files, you are ready to exit the ICU by entering "e <CR>", as shown in Figure B-63.

When the ICU has successfully saved the definition file, it exits after issuing the following message:

```
The Definition File has been written to file:  SAM286.DEF
```

You are now ready to invoke the SUBMIT file SAM286.CSD. Do this by entering

-

The SUBMIT file assembles all the configuration files generated by the ICU and binds the object files with all the libraries required by the subsystems. It then builds the system. Refer again to Figure B-53 for a listing of the output from the SUBMIT file. You may notice warning messages. The warning messages are normal and can be ignored. Only error messages must be heeded. When the SUBMIT file is completed your entire system has been built.

The bootfile named /BOOT/SAM286.286 contains the entire system. You are now ready to bootload your new executable system.

APPENDIX C
PROGRAMMING A 286-BASED SYSTEM INTO PROM DEVICES

C.1 INTRODUCTION

This appendix provides examples of the procedures used to place the iRMX II Operating System into ROM of a 286-based system. All software generation described assumes you are using an Intel System 300 Series Microcomputer. In this appendix, one version of the operating system is created, but two examples of how to program it into ROM exist.

The first example places the iSDM monitor and the iRMX II Bootstrap Loader in ROM along with the generated operating system. A system such as this executes the iSDM monitor code during the initial power-up sequence. This example includes the iSDM monitor and iRMX II Bootstrap Loader for several reasons. First, while you develop the ROM-based system, you can bootstrap load RAM-based versions of the operating system rather than having to switch PROM devices if you are using one processor board. Second, the iSDM monitor, while not allowing breakpoints in PROM DEVICEs, does allow you to disassemble and examine memory in both ROM and RAM. You can use this feature to determine correct code is in correct locations.

The second example eliminates the iSDM monitor and the Bootstrap Loader from the PROM devices. A stand-alone system such as this allows the operating system to be 32 Kbytes larger than in the first example. In this example, the operating system receives control during initial power-up or reset.

C.2 REQUIREMENTS

To use the procedures outlined in this appendix, you must have the following hardware and software:

- A system defined during configuration as residing in ROM.
- The iRMX II.3 and iPPS V1.4 (or newer) software. The iPPS software runs on an Series-IV Development System.

- A 286-based system and a Series-IV Development System connected via the iSDM monitor.
- An iUP-200/201 Universal Programmer with a 'FAST 27/K' module with 27512 support.
- Four 27512 EPROM devices.

C.3 CONFIGURING A ROM-BASED SYSTEM

Before you can program your system into PROM DEVICEs, you must modify a number of parameters in your definition file. These examples assume that you start with the Intel-supplied definition file 28612.DEF located in directory /RMX286/ICU.

To begin the example, you should make a copy of the definition file in a separate directory. To do this, create a new directory in which to do the system generation. The definition file should have a .DEF extension, the boot-loadable system should have a .286 extension, and the system to be loaded into ROM should have a .ROM extension. Entering the three following commands creates a new directory called ROMSYS, attaches you to that directory, and invokes the ICU placing a copy of the definition file 28612.DEF into the new directory. (This example assumes your :HOME: directory is your current default directory.)

```
-
-
-
```

As part of the configuration process, you must perform the following three things in order to fit the iRMX II system developed in these examples into ROM:

- Delete the System Debugger from the system.
- Replace the iRMX II.3 CLI with the iRMX II.1 CLI.
- Delete all Intel device drivers except the 8274 MPSC and MSC device drivers.

Having invoked the ICU, you now can begin to make the necessary configuration changes. Start by modifying the memory screens to define the ROM memory locations the system requires. Do this by first selecting the "Memory for System" screen with the following command:

Entering the previous command causes the "memory for System" screen to appear as follows:

```
(MEMS)          Memory for System

    SYS = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =         010000H,           059FFFH
[2] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
```

Begin the alteration by deleting line 1 by entering the following command:

After the screen reappears, enter the low and high addresses to indicate the location of the system in ROM as follows:

In this example, the system resides in locations 0FC1000H to 0FF7FFFH. These locations allow the iSDM monitor and Bootstrap Loader to reside at locations 0FF8000H to 0FFFFFFH in ROM. After entering the new memory locations, the "Memory for System" screen appears as follows:

```
(MEMS)          Memory for System

    SYS = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =       0FC1000H,           0FF7FFFH
[2] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:<CR>
```

If you were not to include the iSDM monitor and Bootstrap Loader, you could use locations 0FC1000H to 0FFFFEFH.

Now, display the "Memory for Free Space Manager" screen by entering "<CR>", as shown in the previous screen. You must also adjust this memory screen as a ROM-based system uses different memory locations than the RAM-based system defined in the definition file. Change the "Memory for Free Space Manager" screen to contain the memory locations 45000H to 1FFFFFH. The following screen shows the "Memory for Free Space Manager" screen after making the changes:

```
(MEMF)          Memory For Free Space Manager

    FSM = low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM     =       045000H,        01FFFFFH
[2] FSM     =

Enter Changes [Number = new_value / ^D Number / ? / H] :
:
```

After making changes to the memory screens, request the "Sub-systems" screen by entering the following command:

```
(SUB)             Sub-systems

(UDI) Universal Development Interface [Yes/No]    YES
(HI)  Human Interface [Yes/No]                    REQ
(AL)  Application Loader [Yes/No]                 REQ
(RFA) Remote File Access [Yes/No]                 NO
(EIO) Extended I/O System [Yes/No]                REQ
(BIO) Basic I/O System [Yes/No]                   REQ
(SDB) System Debugger [Yes/No]                    YES
(OE)  OS Extension [Yes/No]                       NO

Enter [Abbreviation = new_value / Abbreviation ? / H ] :
:
```

Here, you must delete the System Debugger because of size considerations. Delete the SDB by entering the following command:

Next, request the "Human Interface" screen by entering the following command:

Change the resident initial program (CLI) by entering the following command:

Both the iRMX II.3 CLI and the iRMX II.2 CLI are considerably larger than the iRMX II.1 CLI and cannot be used in this example due to size restrictions.

Next, request the "Intel Device Driver" screen by entering the following command:

Delete all device drivers except the 8274 and MSC drivers by entering the following commands:

```
Do you want any/more iSBC 544A DEVICEs?
```

```
Do you want any/more Terminal Communications Controller Devices?
```

```
Do you want any/more Line Printer for iSBC 286/10 DEVICEs
```

Now, request the "Nucleus" screen by entering the following command:

You must define the number of Global Descriptor Table (GDT) entries your ROM-based system needs. Because ROM space is limited, the number you enter should be the minimum number. Enter the number of real GDT entries required when the system is copied and expanded in RAM using the "ROM Code" screen. Also, because the System Debugger was previously deleted from the configuration, you must change the Default Exception Handler parameter to something other than SDB.

Change the "Nucleus" screen as follows:

```
(NUC)          Nucleus

(NGE) Number of GDT Entries [440 - 8190]                          2000
(NIE) Number of IDT Entries [0-256]                               128
(PV)  Parameter Validation [Yes/No]                               REQ
(ROD) Root Object Directory Size [0 - 3840]                       50
(DEH) Default Exception Handler [Yes/No/SDB/User]                 SDB
(NMI) NMI Exception Handler [Yes/No/Ignore/Job/SDM/User]          IGNORE
(NEB) NMI Enable Byte [0-255]                                     4
(STK) Exception Handler for Stack Exception, Bad TSS and
            Double Fault [Yes/No/Job/SDB]                         JOB
(NEH) Name of Ex Handler Object Module [1-55chs]

(EM)  Exception Mode [Never/Program/Environ/All]                  NEVER
(LSE) Low GDT/LDT Slot Excluded from FSM [440-8189H/None=0]       0
(HSE) High GDT/LDT Slot Excluded from FSM [440-8189/None=0]       0
(RRP) Round Robin Priority Threshold [0-255]                      140
(RRT) Round Robin Time Quota [0-255]                              5
(RIE) Report Initialization Errors [Yes/No]                       YES
(MCE) Maximum Data Chain Elements [0-0FFFFH]                      080H
(CS)  Nucleus Communication Service [Yes/No]                      YES

Enter  [Abbreviation = new_value / Abbreviation ? / H ] :
:nge = 460  <CR>
:deh = job  <CR>
:f rom  <CR>
```

Once you have updated the "Nucleus" screen, view the "ROM Code" screen to change the parameters to match the ROM-based system. Set the RAG parameter to FC1000H to match the starting address defined on the "Memory for System" screen. Always place the GDT at the first memory location reserved for the system. Set the RAS parameter to 2000H to allow enough memory for the controllers in the system. Controllers are always assigned addresses below 2000H. Finally, set the RIA parameter to FC0000H, the starting address of processor board ROM. Change the "ROM Code" screen as follows:

```
(ROM)       ROM code

(SYR) System in ROM [Yes/No]                                    NO
(RAG) ROM Address of Master-GDT [0-0FFFFFFH]                    0H
(NSG) Number Slots in Real GDT [440-8190]                       1024
(RAS) RAM Start Address for System [0-0FFFFFH]                  0H
(RIA) ROM Initialization Code Address [0-0FFFFFFH]              0FF0000H
(RIP) ROM Initialization Procedure [1 - 45 chars]

Enter [Abbreviation = new_value / Abbreviation ? / H ]:
:syr=yes <CR>
:rag=fc1000h <CR>
:ras=02000h <CR>
:ria=fc0000h <CR>
```

Now, request the "Generate File Names" screen by entering the following command:

The "Generate File Names" screen is where you define the pathname of the file containing the ROM-based system. Enter the pathname :$:ROMSYS.ROM as shown below:

```
(GEN)          Generate File Names

File Name [1-55 Characters]

(ROF) ROM Code File Name
                                                    /BOOT/RMX286.ROM
(RAF) RAM Code File Name
                                                    /BOOT/28612.286

Enter  [Abbreviation = new_value / Abbreviation ? / H ] :
:rof=:$:romsys.rom <CR>
:c <CR>
```

Next, enter a "c <CR>" to return to the ICU menu screen.

C.4 GENERATING/BUILDING THE SYSTEM

You are now ready to generate the definition files and build the system. The last step of the previous section caused the ICU main screen to appear. From this screen, enter the Generate (G) command to generate files.

```
 For general help in any screen enter H <CR>.

The following commands are available

Change
Generate
List
Save
Quit
Exit
Replace
Detail-Level
Backup

 ENTER COMMAND :G <CR>
```

After entering "G <CR>" with no prefix, the ICU informs you as each layer is generated (see Figure B-35 for an example). When the system has been generated, the ICU returns to the main menu screen.

Enter the Exit (E) command to write the definition file and exit the ICU as follows:

After entering this command, the ICU informs you that the definition file has been written. It issues the following message before returning control to the command line:

```
The definition file has been written to file:  ROMSYS.DEF
```

You are now ready to invoke the SUBMIT file ROMSYS.CSD that builds the system. The ICU created this SUBMIT file as part of the generation process. Execution of the SUBMIT file generates the application system with the pathname :$:ROMSYS.ROM. Execution of this SUBMIT file also generates the map file ROMSYS.MP2, which contains information crucial to configuring a "tight" system. The map file ROMSYS.MP2 also contains the following warning which you can safely ignore:

```
***  WARNING 119:  SEGMENTS OVERLAP
       LOW ADDRESS:  00FC0000H
       HIGH ADDRESS:  00FC0E04H
```

After the submit file completes execution, use the DOWNCOPY command to copy the resulting file ROMSYS.ROM to the file :F1:ROMSYS.ROM on the hard disk of the Series-IV Development System. This assumes you have used the LNAME command on the Series-IV system to define the logical name :F1 for the directory into which you will copy the file.

Now that the system has been generated, you can either program the PROM devices to include only the operating system by itself (stand-alone) or you can include the iSDM monitor. The next sections describe these different processes.

C.4.1 Including the iSDM™ Monitor and the Bootstrap Loader in the PROM Devices

This section explains how to include the iSDM monitor and iRMX II Bootstrap Loader as part of the operating system that resides in ROM. With this type of system, the iSDM monitor code executes when the system is powered up. In order to create a system that includes the iSDM monitor and the Bootstrap Loader you must prepare two files; one file for each piece of software.

First you must generate a version of the iSDM monitor for the iSBC 286/10(A) board. The iSDM monitor is normally installed on iRMX II systems in the directory /SDM. Since the Bootstrap Loader is to be included in the PROM devices along with the iSDM monitor, be sure you invoke the BOOTSTRAP macro as follows when you configure the iSDM monitor:

The iRMX II.3 iSDM monitor is always located at address 0FF8000H. The iSDM monitor places values into the reset vector and receives control on power-up or reset. Refer to the *iSDM System Debug Monitor User's Guide* for more information on generating the iSDM monitor.

Next, generate a version of the iRMX Bootstrap Loader first stage. Assign your current default directory to be the Bootstrap Loader directory by entering the following command:

Due to the limited amount of space left over for the Bootstrap Loader, you can not leave all available devices selected within BS1.A86. Consequently, you must edit the first stage configuration file, BS1.A86 to contain only the driver included in the ROM system, the MSC device driver. You do not need to change any other parameters in BS1.A86. You must also edit the first stage configuration SUBMIT file (BS1.CSD) to link in only the MSC file.

Execute the Bootstrap Loader SUBMIT file by entering the following command:

-

Complete details on these actions are available in the *Extended iRMX II Bootstrap Loader Reference Manual.*

At this point, the files for the iSDM monitor and the Bootstrap Loader are generated. Copy the resulting files, BS1 and 28610A, to the files :F1:BS1 and :F1:28610A of the hard disk of the Series-IV Development System using the DOWNCOPY command. This example assumes you have used the LNAME command on the Series-IV system to define the logical name :F1: as the directory into which you will copy the files. The following commands move the two files:

-
-

The next six sections describe how to program the PROM devices.

C.4.1.1 Setting Up the iUP 201 PROM Programmer

Perform the following three steps to set up the iUP 201 PROM Programmer:

1. Make sure that the RS-232A line is connected from the iUP 201 programmer to the Series-IV system.
2. Insert the 'FAST 27K' module into the iUP 201 28-pin socket and turn on the power to the iUP 201 Universal Programmer.
3 Press the ONLINE button on the iUP 201 front panel.

WARNING

While following the steps outlined in this section, you must closely adhere to any warnings or cautions given in the *iUP-200/201 Universal Programmer User's Guide.*

C.4.1.2 Formatting the Operating System PROM File

Invoke the iPPS software and issue the following commands to format the two PROM image files. Because this example includes the iSDM monitor and Bootstrap Loader as part of the operating system, the memory range for the operating system goes from OFC0000H to 0FF7FFFH. If you were programming only the operating system, the largest memory range for it would range from 0FC0000H to 0FFFFEFH.

```
- IPPS  <CR>

INTEL PROM PROGRAMMING SOFTWARE, Vx.y
COPYRIGHT INTEL CORPORATION years

PPS>type 27512  <CR>
PPS>initialize 286  <CR>
PPS>format :f1:romsys.rom (0FC0000h, 0FF7FFFh)  <CR>
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N_BYTE=4)
LU =3  <CR>
INPUT BLOCK SIZE (N BYTES)
N =2  <CR>
OUTPUT BLOCK SIZE (N BYTES)
N =1  <CR>

INPUT BLOCK STRUCTURE.
NUMBER OF INPUT LOGICAL UNITS =002

LSB
-------
|00|01|
-------
NUMBER OF OUTPUT LOGICAL UNITS =001

OUTPUT SPECIFICATION (<CR> TO EXIT):
*0 to :f1:romsys.evn  <CR>
OUTPUT STORED
*1 to :f1:romsys.odd  <CR>
OUTPUT STORED
* <CR>
PPS>
```

The parameters in parenthesis on the FORMAT line tell the IPPS software what memory range of code you wish to program into PROM devices.

C.4.1.3 Programming the Operating System into PROM Devices

To actually place the generated system into the PROM devices, begin by inserting the first PROM device into the active socket of the PROM programmer. Next, enter the following PPS commands shown as bolded text:

```
PPS>copy :f1:romsys.evn to prom  <CR>
----CAUTION----PROGRAMMING THE FULL LENGTH REQUIRES MORE THAN ONE
                PROM.
 CHECKSUM=value
FIRST INSTALL THE NEW/NEXT PROM AND THEN CONTINUE.
 CONTINUE--Y/N? Y
 CHECKSUM=value
PPS>exit  <CR>
```

During the copy, the IPPS software automatically determines that the contents of the file require more than one of the specified type of PROM devices to contain the code. When a new PROM device is needed, IPPS prompts you with the following message:

```
INSTALL THE NEW/NEXT PROM AND THEN CONTINUE
```

Remove the first PROM device and insert the second. Next, type "Y <CR>". When the checksum value appears on the screen, remove the second PROM device. These two PROM devices you have removed contain the even (or low) bytes of the WORD values that compose the operating system. As you remove the PROM devices from the programmer, carefully label them; unlabeled PROM devices look very much alike. At a later time, you will place the first programmed PROM device in socket U41 and the second programmed PROM device in socket U40 of an iSBC 286/10(A) board. (Use sockets U2 and U3 of the iSBX 341 on an iSBC 286/12 board.)

Next, you need to program the PROM devices to contain the odd (or high) bytes of the WORD values that compose the operating system. Insert the third PROM device into the active socket of the PROM programmer and enter the following PPS commands shown as bolded text:

```
PPS>copy :f1:romsys.odd to prom  <CR>
----CAUTION----PROGRAMMING THE FULL LENGTH REQUIRES MORE THAN ONE
                PROM.
 CHECKSUM=value
FIRST INSTALL THE NEW/NEXT PROM AND THEN CONTINUE.
 CONTINUE--Y/N? Y
 CHECKSUM=value
PPS>
```

Again, when IPPS determines it needs another PROM device, it prompts you with the following message:

```
INSTALL THE NEW/NEXT PROM THEN CONTINUE
```

Remove the third PROM device and insert the fourth. Next, type "Y <CR>". When the checksum value appears on the screen, remove the fourth PROM device. These two PROM devices contain the odd (or high) bytes of the WORD values that compose the system. As you remove the PROM devices from the programmer, carefully label them; unlabeled PROM devices look very much alike.. At a later time, you will place the third programmed PROM device in socket U76 and the fourth programmed PROM device in socket U75 of an iSBC 286/10(A) board. (Use sockets U5 and U6 of the iSBX 341 on an iSBC 286/12 board.) The next step programs the iSDM monitor and the iRMX Bootstrap Loader into the fourth PROM device so do not install it now.

C.4.1.4 Programming the iSDM Monitor into PROM Devices

This section describes how to program the iSDM monitor into the second and fourth PROM devices. To do this, insert the second PROM device and enter the following commands shown as bolded text:

```
PPS>format :f1:28610a (0FF8000h, 0FFFFFFh)  <CR>
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N_BYTE=4)
LU =3  <CR>
INPUT BLOCK SIZE (N BYTES)
N =2  <CR>
OUTPUT BLOCK SIZE (N BYTES)
N =1  <CR>

OUTPUT SPECIFICATION (<CR> TO EXIT):
*0 to :f1:28610a.evn  <CR>
OUTPUT STORED
*1 to :f1:28610a.odd  <CR>
OUTPUT STORED
<CR>
PPS>

PPS>copy :f1:28610a.evn to prom(0C000H)  <CR>
 CHECKSUM=value
```

When the IPPS software prompts you with the following message, remove the second PROM device, insert the fourth, and enter the following command:

```
PPS>copy :f1:28610a.odd to prom(0C000H)  <CR>
 CHECKSUM=value
```

C.4.1.5 Programming the Bootstrap Loader into PROM Devices

This section describes how to program the Bootstrap Loader into the second and fourth PROM devices. To begin, insert the second PROM device and enter the commands shown as bolded text:

```
PPS>initialize 86  <CR>
PPS>format :f1:BS1 (0FE400h, 0FFF7Fh)  <CR>
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N_BYTE=4)
LU =3  <CR>
INPUT BLOCK SIZE (N BYTES)
N =2  <CR>
OUTPUT BLOCK SIZE (N BYTES)
N =1  <CR>

OUTPUT SPECIFICATION (<CR> TO EXIT):
*0 to :f1:bs1.evn  <CR>
OUTPUT STORED
*1 to :f1:bs1.odd  <CR>
OUTPUT STORED
<CR>
PPS>

PPS>copy :f1:bs1.evn to prom(0F200H)  <CR>
 CHECKSUM=value
```

When the IPPS software displays the following prompt, remove the second PROM device, insert the fourth PROM device, and enter the following command:

```
PPS>copy :f1:bs1.odd to prom(0F200H)  <CR>
 CHECKSUM=value
PPS>exit  <CR>
```

C.4.1.6 Starting the Operating System in ROM from the iSDM™ Monitor

The four PROM devices now contain the operating system, the iSDM monitor, and the Bootstrap Loader. Perform the following steps to start the system:

1. Place the first programmed PROM device in socket U41 of the iSBC 286/10(A) board.
2. Place the second programmed PROM device in socket U40 of the iSBC 286/10(A) board.
3. Place the third programmed PROM device in socket U76 of the iSBC 286/10(A) board.
4. Place the fourth programmed PROM device in socket U75 of the iSBC 286/10(A) board.

5. Insert the iSBC 286/10(A) board into the system chassis and apply power to the hardware.
6. Enter the following command from the iSDM monitor to activate the iRMX II system:

.

The code executed by entering the command in step 6 above is the ROM initialization procedure whose address was specified in the ICU as 0FC0000H. The offset of the beginning instruction of this procedure is 12H. The reason 0C000:12 is used instead of 0FC000:12 is that, in real address mode, the 80286 can only access up to 1 megabyte of memory. Also, in real address mode, the memory decoding of the 286/10(A) board, when jumpered for 27512 EPROM devices, places the memory addresses of the PROM devices at 0C000:0 to 0FFFF:F. This addressing scheme results in execution of the code at 0C000:12 to occur first. The code at 0C000:12 is the first instruction of the ROM initialization code.

C.4.2 Creating a System That is Activated on Power-Up

The previous section explained how to program a system that included the operating system, the iSDM monitor, and the Bootstrap Loader into PROM devices. With this type of a system, the iSDM monitor receives control during the initial power-on (or reset) sequence and is signed on. This requires that you give a command to the iSDM monitor to start the operating system. This section explains how to program only the operating system into the PROM devices. To create a PROM-based iRMX II system which receives control at power-on, you need to take certain steps. Before the following section describes these steps, you should be made aware of the following:

- Most PROM-based systems are already "debugged" and have no need of the iSDM monitor or the iRMX II System Debugger (SDB).
- The iSDM monitor has limited use in a PROM-based system because it can't set breakpoints in code that executes from ROM.
- The ROM initialization code of the iRMX II Nucleus currently has no means of fully initializing the iSDM monitor. Initialization code can only inform the iSDM monitor (if it is present and has been previously initialized) that the 80286 CPU has been switched to Protected Mode.
- Virtually all PROM-based iRMX II systems require that the system be initialized immediately upon Power Up.

For these reasons, a standalone PROM-based iRMX II Operating System has no reason to include the iSDM monitor, the Bootstrap Loader or the SDB. With this in mind, the following sections explain how to program such a system into the PROM devices and detail the additional steps required to manually program the Reset Vector. (You must manually program the Reset Vector in order to have the operating system receive control after the initial power-on or reset sequence.)

C.4.2.1 Formatting the Operating System PROM File

To begin, you must prepare two PROM image files: one for the even bytes and one for the odd bytes of the operating system. To prepare these files, invoke the iPPS software and issue the following commands shown as bolded text. Because you are now programming only the operating system into the PROM devices, the largest possible memory range for the system ranges from 0FC0000H to 0FFFFEFH. The parameters in parenthesis on the FORMAT line tell the IPPS software what memory range of code you wish to program into the PROM devices. If your system does not extend to 0FFFFEFH, you only need to specify the address at which it does reach, although there is usually no harm in specifying the entire range.

The following screen shows the commands necessary and the response you receive at the terminal:

```
- IPPS  <CR>

INTEL PROM PROGRAMMING SOFTWARE, Vx.y
COPYRIGHT INTEL CORPORATION years

PPS>type 27512  <CR>
PPS>initialize 286  <CR>
PPS>format :f1:romsys.rom (0FC0000h, 0FFFFEFh)  <CR>
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N_BYTE=4)
LU =3  <CR>
INPUT BLOCK SIZE (N BYTES)
N =2  <CR>
OUTPUT BLOCK SIZE (N BYTES)
N =1  <CR>

OUTPUT SPECIFICATION (<CR> TO EXIT):
*0 to :f1:romsys.evn  <CR>
OUTPUT STORED
*1 to :f1:romsys.odd  <CR>
OUTPUT STORED
* <CR>
PPS>
```

C.4.2.2 Programming the Operating System Into the PROM Devices

This section describes how to program the two operating system files prepared in the previous section into the PROM devices and how to modify the Reset Vector jump address.

Two 27512 PROM devices are required to contain the even bytes of the operating system and two 27512 PROM devices are required to contain the odd bytes of the operating system. This example refers to these PROM devices as one through four.

You can program the first of these PROM devices with no intermediate steps. Begin by inserting the first PROM device into the active socket and enter the following command:

```
PPS>copy :f1:romsys.evn(0,0FFFFH) to prom  <CR>
 CHECKSUM=value
```

After the CHECKSUM = value message appears, remove the first PROM device, insert the second PROM device, and enter the following command:

```
PPS>copy :f1:romsys.evn(10000H,01FFFFH) to buffer  <CR>
 CHECKSUM=value
```

You now need to modify the even bytes of the Reset Vector jump address. Use the iPPS SUBSTITUTE command to change the even bytes of the Reset Vector in the buffer as follows:

00FFF8: FF FF FF FF FF FF FF FF

to

00FFF8: EA 00 C0 FF FF FF FF FF

After changing the values, copy the buffer to a file by entering the following command:

```
- copy buffer to :f1:temp
```

Now, copy the temp file to the PROM devices by entering the following command:

```
PPS>copy :f1:temp to prom  <CR>
 CHECKSUM=value
```

After the CHECKSUM=value message appears on the screen, remove the second PROM device. The first two PROM devices now contain the even (or low) bytes of the WORD values that compose the operating system. As you remove the PROM devices from the programmer, carefully label them; unlabeled PROM devices look alike. Place the first programmed PROM device in socket U41 and the second programmed PROM device in socket U40 of an iSBC 286/10(A) board.

You can program the third and fourth PROM devices with no intermediate steps also. Begin by inserting the third PROM device into the active socket and enter the following command:

```
PPS>copy :f1:romsys.odd(0,0FFFFH) to prom  <CR>
 CHECKSUM=value
```

After the CHECKSUM=value message appears, remove the third PROM device, insert the fourth PROM device, and enter the following command:

```
PPS>copy :f1:romsys.odd(10000H,01FFFFH) to buffer  <CR>
 CHECKSUM=value
```

You now need to modify the odd bytes of the Reset Vector jump address. Use the iPPS SUBSTITUTE command to change the odd bytes of the Reset Vector in the buffer as follows:

00FFF8: EA 00 C0 FF FF FF FF FF

to

00FFF8: 12 00 FF FF FF FF FF FF

After changing the values, copy the buffer to a temporary file by entering the following command:

```
- copy buffer to :f1:temp
```

Now, copy the temporary file to the PROM devices by entering the following command:

```
PPS>copy :f1:temp to prom  <CR>
 CHECKSUM=value
```

After the CHECKSUM = value message appears on the screen, remove the fourth PROM device. The third and fourth PROM devices now contain the odd (or high) bytes of the WORD values that compose the operating system. As you remove the PROM devices from the programmer, carefully label them; unlabeled PROM devices look alike. Place the third programmed PROM device in socket U76 and the fourth programmed PROM device in socket U75 of an iSBC 286/10(A) board.

The iPPs SUBSTITUTE commands above provide a FAR JUMP instruction to the ROM initialization procedure whose address is specified in the ICU as 0FC0000H. The offset of the beginning instruction of this procedure is 12H. The reason 0C000:12 is used instead of 0FC000:12 is that, in real address mode, the 80286 can only access up to 1 megabyte of memory. Also, in real address mode, the memory decoding of the 286/10(A) board, when jumpered for 27512 EPROM devices, places the memory addresses of the PROM devices at 0C000:0 to 0FFFF:F. This addressing scheme results in execution of the code at 0C000:12 to occur first. The code at 0C000:12 is the first instruction of the ROM initialization code.

C.4.2.3 Starting the Operating System in ROM

The four PROM devices now contain the operating system and the altered Reset Vector jump address. Perform the following steps to start the system:

1. Place the first programmed PROM device in socket U41 of the iSBC 286/10(A) board.
2. Place the second programmed PROM device in socket U40 of the iSBC 286/10(A) board.
3. Place the third programmed PROM device in socket U76 of the iSBC 286/10(A) board.
4. Place the fourth programmed PROM device in socket U75 of the iSBC 286/10(A) board.
5. Insert the iSBC 286/10(A) board into the system chassis and apply power to the hardware. The iRMX II system is immediately initialized and sign on occurs using the terminal(s) specified in :CONFIG:TERMINALS on the system device.

C.4 HARDWARE JUMPER MODIFICATIONS

To program the system into PROM devices as in the above example, the following jumpers were changed on the iSBC 286/10(A) board:

To specify 4 27512 EPROM devices, set up jumpers 62 through 91 as follows;

Default Configuration	Jumpers to Set for 27512 EPROMS
E62 - E63	E65 - E67
E70 - E72	E68 - E70
E71 - E73	E71 - E73
E75 - E76	E75 - E76
E77 - E78	E80 - E82
E85 - E87	E83 - E85
E86 - E88	E86 - E88
E90 - E91	E90 - E91

To specify a starting memory address and memory size for local memory, use primary decode option 3. The jumpers required are

Default Configuration	Jumpers for Primary Decode Option 3
E218 - E219 installed	E218 - E219 installed
E220 - E221 removed	E220 - E221 installed

To specify memory/size/justification for local memory, use secondary option 3. The jumpers required are

Default Configuration	Jumpers for Secondary Option 3
E51 - E59 removed	E51 - E59 removed
E50 - E58 removed	E50 - E58 installed
E49 - E57 installed	E49 - E57 installed

APPENDIX D
PROGRAMMING A 386/100-BASED SYSTEM INTO PROM DEVICES

D.1 INTRODUCTION

This appendix provides an example of the procedures used to place the iRMX II Operating System into ROM for a 386/100-based system. All software generation described assumes you are using an Intel System 300 Series Microcomputer. In addition, because a Series-IV Development System is required to run the PROM programmer, the iSDM monitor is used to provide the serial link between the iRMX system and the Series-IV system. In this example, one version of the operating system is created.

The example consists of three processes: configuring a ROM-based system, generation/building of the system, and actually placing the system into ROM. In this example, the iSDM monitor and Bootstrap Loader are not included as part of the generated system. This allows the operating system to be 32 Kbytes larger. For a system generated in this manner, the operating system receives control on power-on or reset.

The last section of this appendix describes the configuration differences existing when programming a 386/100-based system and a 386/20-based system into PROM devices.

D.2 REQUIREMENTS

To use the procedures outlined in this appendix, you must have the following hardware and software:

- A system defined during configuration as residing in ROM.
- The iRMX II.3 and iPPS V1.4 (or newer) software. The iPPS software runs on an Series-IV Development System.
- A 386/100-based system and a Series-IV Development System connected via the iSDM monitor.
- An iUP-200/201 Universal Programmer with a 'GUPI 27010' module.
- Two 27010 EPROM devices.

D.3 CONFIGURING A ROM-BASED SYSTEM

Before you can program your system into ROM, you must modify a number of parameters in your definition file. These examples assume that you start with the Intel-supplied definition file 386100.DEF located in directory /RMX286/ICU.

To begin the example, you should make a copy of the definition file in a separate directory. To do this, create a new directory in which to do the generation. Give the directory the same name as the definition file and the output system. The definition file should have a .DEF extension, the boot-loadable system should have a .286 extension and the system to be loaded into PROM devices should have a .ROM extension. Entering the three following commands creates a new directory called ROMSYS, attaches you to that directory, and invokes the ICU placing a copy of the definition file 386100.DEF into the new directory. (This example assumes your :HOME: directory is your current default directory.)

```
-
-
-
```

As part of the configuration process, you must perform the following two things in order to fit the iRMX II system developed in this example into PROM devices.

- Delete the System Debugger and UDI from the system.
- Replace the iRMX II.3 CLI with the iRMX II.1 CLI.

Having invoked the ICU, you now can begin to make the necessary configuration changes. Start by modifying the memory screens to define the ROM memory locations the system requires. Do this by first selecting the "Memory for System" screen with the following command:

Entering the previous command causes the "memory for System" screen to appear as follows:

```
(MEMS)          Memory for System

    SYS = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS     =         010000H,           059FFFH
[2] SYS     =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
```

At this screen, you must configure the amount of memory required for the generated system. When a 386-based system is initialized, it moves from ROM to RAM and then begins execution. The area in RAM where it resides is the lowest block of memory specified in the "Memory for System" screen. Thus, you must specify the lowest block of memory with enough space to fit the system. At this point, because you do not know the exact size needed, you would normally supply a guessed value to replace 06FFFEH. Then, after the real system is generated, you could determine the actual size using information given in the .MP2 file created at generation time. For this example, however, the amount of memory needed is known beforehand as 10000H to 4FF00H.

Configure the memory needed for the system using the following two commands. These commands delete the default information and add the new information.

After entering the memory locations, the "Memory for System" screen appears as follows:

```
(MEMS)          Memory for System

    SYS = low [0-0FFFFFFH], high [0-0FFFFFFH]
[1] SYS   =       010000H,          4FF00H
[2] SYS   =

Enter Changes [Number = new_value / ^D Number / ? / H ] :
:<CR>
```

Now, display the "Memory for Free Space Manager" screen by entering "<CR>", as shown in the previous screen. Because a ROM-based system uses different memory locations than a RAM-based system, you must adjust this memory screen. The memory for the Free Space Manager must not overlay system memory or the writeable data segments, which reside just above system memory. To ensure that you specify the correct starting address for the Free Space Manager memory, perform the following steps:

1. Calculate the sum of the read/write segment limits found in the .MP2 file.
2. Find the segment with the largest end address.
3. Add together the result of step one and the address found in step two.
4. Use the result of step three for the minimum start address for the Free space manager.

After determining the start address for the Free Space Manager, make the changes by entering in commands similar to the ones you entered to change the "Memory for System" screen. The only difference is use memory locations 65000H to 7FFFFFH. The following screen shows the "Memory for Free Space Manager" screen after making changes.

```
(MEMF)          Memory For Free Space Manager

    FSM = low [0-0FFFFFFH], high [0-0FFFFFH]
[1] FSM   =       65000H,          7FFFFFH
[2] FSM   =

Enter Changes [Number = new_value / ^D Number / ? / H] :
:f sub  <CR>
```

After making changes to the memory screens, request the "Sub-systems" screen using the command "f sub <CR>" shown in the previous screen.

```
(SUB)            Sub-systems

(UDI) Universal Development Interface [Yes/No]    YES
(HI)  Human Interface [Yes/No]                    REQ
(AL)  Application Loader [Yes/No]                 REQ
(RFA) Remote File Access [Yes/No]                 NO
(EIO) Extended I/O System [Yes/No]                REQ
(BIO) Basic I/O System [Yes/No]                   REQ
(SDB) System Debugger [Yes/No]                    YES
(OE)  OS Extension [Yes/No]                       NO

Enter [Abbreviation = new_value / Abbreviation ? / H ] :
: sdb=n <CR>
: udi=n <CR>
: hi=y <CR>
: f hi <CR>
```

Here, you must delete the System Debugger and the UDI because of size considerations. You also select the Human Interface for your system by setting answering yes to "hi". The last command above requests the "Human Interface" screen

Change the resident initial program (CLI) versions by entering the following command:

Both the iRMX II.3 CLI and the iRMX II.2 CLI are considerably larger than the iRMX II.1 CLI and cannot be used in this example due to size restrictions.

Next, request the "Nucleus" screen by entering the following command:

You now must define the number of Global Descriptor Tables (GDT) entries your ROM-based system needs. Because ROM space is limited, the number you enter should be the minimum number of entries needed. The number of real GDT entries required when the system is copied and expanded in RAM is entered on the "ROM Code" screen.

You also must change the Default Exception Handler parameter to something other than SDB since the SDB was previously deleted from the configuration.

Enter the following commands to make the necessary changes to the "Nucleus Screen":

After making these changes, the "Nucleus Screen" appears as follows:

```
(NUC)          Nucleus

(NGE) Number of GDT Entries [440 - 8190]                          460
(NIE) Number of IDT Entries [0-256]                               128
(PV)  Parameter Validation [Yes/No]                               REQ
(ROD) Root Object Directory Size [0 - 3840]                       50
(DEH) Default Exception Handler [Yes/No/Job/SDB/User]             JOB
(NMI) NMI Exception Handler [Yes/No/Ignore/Job/SDM/User]          IGNORE
(NEB) NMI Enable Byte [0-255]                                     4
(STK) Exception Handler for Stack Exception, Bad TSS and
            Double Fault [Yes/No/Job/SDB]                         JOB
(NEH) Name of Ex Handler Object Module [1-55chs]

(EM)  Exception Mode [Never/Program/Environ/All]                  NEVER
(LSE) Low GDT/LDT Slot Excluded from FSM [440-8189H/None=0]       0
(HSE) High GDT/LDT Slot Excluded from FSM [440-8189/None=0]       0
(RRP) Round Robin Priority Threshold [0-255]                      255
(RRT) Round Robin Time Quota [0-255]                              5
(RIE) Report Initialization Errors [Yes/No]                       YES
(MCE) Maximum Data Chain Elements [0-FFFFH]                       080H
(CS)  Nucleus Communication Service                               YES

Enter  [Abbreviation = new_value / Abbreviation ? / H ] :
```

After updating the "Nucleus" screen, enter the the following command to view the "ROM Code" screen:

The "f rom <CR>" command causes the "ROM Code" screen to appear as follows:

```
(ROM)       ROM code

(SYR) System in ROM [Yes/No]                                    NO
(RAG) ROM Address of Master-GDT [0-0FFFFFFH]                    0H
(NSG) Number Slots in Real GDT [440-8190]                       1024
(RAS) RAM Start Address for System [0-0FFFFFH]                  0H
(RIA) ROM Initialization Code Address [0-0FFFFFFH]              0FF0000H
(RIP) ROM Initialization Procedure [1 - 45 chars]

Enter [Abbreviation = new_value / Abbreviation ? / H ]:
```

Use the following command to inform the ICU that your system is to reside in ROM:

The RAG field provides the 24-bit physical address of the master Global Descriptor Table (GDT) as it is to be burned into ROM. For a 386/100-based system, RAG also indicates the starting address of ROM code and data that is copied into RAM. For this reason, you should locate the GDT at the start of ROM. In this example, we are using 27010 EPROM devices so you should set the RAG field to 0FC0000H using the following command:

The RAS field gives the starting address where the nucleus initialization code is to begin copying writeable segments of the system into RAM. Consequently, you must ensure that the writeable blocks do not overlay system memory specified in the MEMS screen or the memory managed by the free space manager specified in the MEMF screen. In this example, the system is copied to RAM starting at address 010000H to approximately 04FF00H. Thus to place the writeable segments above the executable code, assign RAS to 04FF00H using the following command:

The RIA determines the 24 bit address of the nucleus initialization code. This address must be located in the first megabyte of RAM. Normally, you would assign RIA to 0 when you build the system the first time. After the system build, you could then examine the .MP2 file and see where the builder located the initialization code. Having obtained the address the builder used, you would then Assign the base address to RIA and generate the system again. By relying on the builder to locate RIA, memory fragmentation is minimized. Note, a system generated with zero (0) assigned to RIA does not work. For this example, however, assign 013A30H to RIA using the following command:

Enter a carriage return <CR> and verify that the "ROM Code" screen appears as follows:

```
(ROM)       ROM code

(SYR) System in ROM [Yes/No]                                    YES
(RAG) ROM Address of Master-GDT [0-0FFFFFFH]                    FC0000H
(NSG) Number Slots in Real GDT [440-8190]                       1024
(RAS) RAM Start Address for System [0-0FFFFFH]                  4FF00H
(RIA) ROM Initialization Code Address [0-0FFFFFFH]              013A30H
(RIP) ROM Initialization Procedure [1 - 45 chars]

Enter [Abbreviation = new_value / Abbreviation ? / H ]:
```

Now, request the "Generate File Names" screen by entering the following command:

Entering the above command causes the "Generate File Names" screen to appear as below:

```
(GEN)          Generate File Names

File Name [1-55 Characters]

(ROF) ROM Code File Name
                                                    /BOOT/RMX286.ROM
(RAF) RAM Code File Name
                                                    /BOOT/386100.286

Enter  [Abbreviation = new_value / Abbreviation ? / H ]:
```

The "Generate File Names" screen is where you define the pathname of the file containing the ROM-based system. Specify the pathname by entering the following command:

After making the change, the screen looks as follows:

```
(GEN)          Generate File Names

File Name [1-55 Characters]

(ROF) ROM Code File Name
                                                    :$:ROMSYS.ROM
(RAF) RAM Code File Name
                                                    /BOOT/386100.286

Enter  [Abbreviation = new_value / Abbreviation ? / H ] :c <CR>
```

D.4 GENERATING/BUILDING THE SYSTEM

You are now ready to generate the definition files and build the system. Entering a "C" as the last step in the previous section causes the ICU main screen to appear. From this screen, enter the Generate (G) command to generate files.

```
For general help in any screen enter H <CR>.

   The following commands are available

   Change
   Generate
   List
   Save
   Quit
   Exit
   Replace
   Detail-Level
   Backup

    ENTER COMMAND :G <CR>

     ENTER a letter to be used as prefix :<CR>
```

After entering "G <CR>", the ICU asks you to enter a single letter to be used as a prefix to the generated file names. Entering a carriage return causes no prefix to be used. After responding to the prefix prompt, the ICU informs you of each layer as it is generated (see Figure B-35 for an example). When the system has been generated, the ICU returns to the main menu screen.

Enter the Exit (E) command to write the definition file and exit the ICU as follows:

After entering this command, the ICU informs you that the definition file has been written. It issues the following message before returning control to the command line:

```
The definition file has been written to file:  ROMSYS.DEF
```

You are now ready to invoke the SUBMIT file ROMSYS.CSD that builds the system. The ICU created this SUBMIT file as part of the generation process. Execution of the SUBMIT file generates the application system with the pathname :$:ROMSYS.ROM. Execution of this SUBMIT file also generates the map file ROMSYS.MP2, which contains information crucial to configuring a "tight" system. The map file ROMSYS.MP2 also contains the following warnings which you can safely ignore:

```
***  WARNING 269:  LINE 22, NEAR CLI_DATA: Segment Size Reduced
```

```
***  WARNING 119:  SEGMENTS OVERLAP
       LOW ADDRESS:  FFFFA0H
       HIGH ADDRESS:  FFFFF2H
```

After the submit file completes execution, use the DOWNCOPY command to copy the resulting file ROMSYS.ROM to the file :F1:ROMSYS.ROM on the hard disk of the Series-IV Development System. This assumes you have used the LNAME command on the Series-IV system to define the logical name :F1 for the directory into which you will copy the file.

Now that the system has been generated, you must program the PROM devices to include the stand-alone system.

D.5 PROGRAMMING THE SYSTEM INTO PROM DEVICES

This section explains how to program the operating system into PROM devices. To create a PROM-based iRMX II system which receives control at power-on, you need to take certain steps. Before the following section describes these steps, you should be made aware of the following:

- Most PROM-based systems are already "debugged" and have no need of the iSDM monitor or the iRMX II System Debugger (SDB).

- The iSDM monitor has limited use in a PROM-based system because it can't set breakpoints in code that executes from ROM.
- The ROM initialization code of the iRMX II Nucleus currently has no means of fully initializing the iSDM monitor. Initialization code can only inform the iSDM monitor (if it is present and has been previously initialized) that the 80286 CPU has been switched to Protected Mode.
- Virtually all PROM-based iRMX II systems require that the system be initialized immediately upon Power Up.

For these reasons, a standalone PROM-based iRMX II Operating System has no reason to include the iSDM monitor, the Bootstrap Loader, or the SDB. With this in mind, the following sections explain how to program such a system into PROM devices.

D.5.1 Formatting the Operating System PROM File

Invoke the iPPS software and issue the following commands (user input is in bold) to prepare the four PROM image files. The parameters in parenthesis on the FORMAT line tell the IPPS software what memory range of code you wish to program into PROM devices.

```
- IPPS  <CR>

INTEL PROM PROGRAMMING SOFTWARE, Vx.y
COPYRIGHT INTEL CORPORATION years

PPS>type 27010  <CR>
PPS>initialize 286  <CR>
PPS>format :f1:romsys.rom (010000h, 04ff00h)  <CR>
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N_BYTE=4)
LU =3  <CR>
INPUT BLOCK SIZE (N BYTES)
N =2  <CR>
OUTPUT BLOCK SIZE (N BYTES)
N =1  <CR>

INPUT BLOCK STRUCTURE.
NUMBER OF INPUT LOGICAL UNITS = 002

LSB
-------
|00|01|
-------

NUMBER OF OUTPUT LOGICAL UNITS = 001

OUTPUT SPECIFICATION (<CR> TO EXIT):
*0 to :f1:romsys.evn  <CR>
OUTPUT STORED
*1 to :f1:romsys.odd  <CR>
OUTPUT STORED
* <CR>
PPS>
```

D.5.2 Formatting the Copy Routine

This step involves formatting the copy routine that actually executes in ROM. Enter the following commands shown in bold text to perform this task:

```
PPS>format :f1:romsys.rom(0ffffa0h)
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N_BYTE=4)
LU =3  <CR>
INPUT BLOCK SIZE (N BYTES)
N =2  <CR>
OUTPUT BLOCK SIZE (N BYTES)
N =1  <CR>

OUTPUT SPECIFICATION (<CR> TO EXIT):
*0 to :f1: copyrom.evn <CR>
OUTPUT STORED
*1 to :f1:copyrom.odd <CR>
* <CR>
```

D.5.3 Copying the Operating System and the Copy Routing to ROM

To copy the system and the copy routine to ROM, you must copy even and odd bytes of the system and the copy routine using the PPS copy command. First insert a PROM device and use the PPS copy command to burn the even bytes of the system and the copy routine into the PROM devices:

```
PPS> copy romsys.evn to PROM
CHECKSUM = VALUE
PPS> copy copyrom.evn to PROM(1ffd0h)
CHECKSUM = VALUE
```

Now you must burn in the odd bytes of the system and the copy routine. Insert the remaining PROM device and issue the following two commands:

```
PPS> copy romsys.odd to PROM
CHECKSUM = VALUE
PPS> copy copyrom.odd to PROM(1ffd0h)
CHECKSUM = VALUE
```

Notice that the copy command for the copy routine is entered in the following form:

copy copyrom.xxx to PROM(offset)

The value offset causes the copy routine to be physically placed away from the system. To determine the value for offset use the following formula:

offset = (start_address_of_copy_routine - PROM_start_address) / 2

For this example, offset is calculated as follows:

offset = (FFFFA0H - FC0000H) / 2 = 1FFD0H

D.6 BOOTING THE SYSTEM

After placing the system into PROM devices, you can boot the system. In this example, the system included the Human Interface subsystem. Consequently, the system must also include a system device (hard disk drive or flexible diskette drive). If your system does not include the Human Interface, the system does not have to include a system device. For systems that include a system device, you must perform a system reset after initially supplying power to the system. The reason for performing a reset after power-up is because the system device needs time to reach maximum spinning speed before the Human Interface can successfully access files on the device.

D.7 CONSIDERATIONS FOR A 386/20-BASED SYSTEM

If the system you are building is going to run on a 386/20-based machine, some configuration differences exist. The following list summarizes these differences:

- Because ROM is mapped to the lower megabyte of memory when the system is in real mode, you must supply a different ROM address for the master Global Descriptor Table in the "ROM Code" screen. Assuming your system uses 27010 EPROM devices, supply a value of 0C0000H for the RAG field in the "ROM Code" screen during configuration. This value causes the GDT to be placed at the start of ROM.
- You must configure the Initialize On-board Functions (IF) field in the "Hardware" screen to 3 rather than 4. The value 3 indicates the board being initialized is an iSBC 386/20 board.

INDEX

A

Abbreviated screen names 1-18
Aborting ICU commands 1-24
Access rights to definition files 1-7
Adding a RAM driver B-21
Adding an Intel-device driver B-15
Adding an Intel-supplied controller B-6
Adding new device drivers 1-45, 46, 47
Adding unit information B-10, 18
Adding users to your system 4-1
Application code 3-1
Application jobs 3-1
ASCII backup files 1-23
Assembling configuration files 2-4, B-34, D-10

B

Backup command 1-23
Binding application jobs 3-4, 7
Binding the subsystems 2-4, B-34, C-8, D-10
BND286 3-4
Bootstrap loading a system B-40
Bootstrap Loader inclusion C-9
Build files 1-18
Building application jobs 3-4, 7
Building the system 2-5, B-34, C-8, D-10

C

Change command 1-16, 25, B-41, C-2, D-2
Changing definition files 1-17, 29
Changing the editing control character 1-21
Changing system memory B-29, C-2, D-2
Choosing a definition file 1-7
Code segment size 3-8
Command mode 1-14

Commands
Backup 1-23
Change 1-16, B-41, C-2, D-2
Detail-level 1-22
Exit 1-21, 2-1, B-34
Generate 1-18, 2-1, B-32, 44, C-8, D-9
Help 1-15
List 1-19, 2-1, B-44
Quit 1-20
Replace 1-21
Save 1-19, 2-1
Communication board 3-3
Configuration
Environments 1-4
Files 2-2, 3, A-1
Generating files 2-1
Configuring users into your system 4-1
Control-C 1-24
Conventions iv
Copy routine D-13
Copying the current screen 1-33, 34
Creating build and submit files 1-18
Creating directories for your systems 1-10, C-2, D-2
Creating systems that activate on power-up C-16

D

Data segment size 3-8
Debugging application jobs 5-6
Definition file 1-1, 7, B-1, C-8, D-10
Deleting a repetitive-fixed screen 1-38
Deleting a screen 1-33, 38
Deleting an element 1-33, 34, 35
Deleting data on a repetitive screen format 1-35
Determining memory locations 3-3, 6
Displaying the next screen 1-33, 34
Displaying the previous screen 1-33

E

Editing a screen 1-29, 32, B-6, 7
Editing control character 1-21
Editing definition files 1-17, 25
Elements of a screen 1-26
Eliminating excess space 3-6, B-41

Ending an ICU session 1-20, 21
Errors
Assembling the configuration files 2-6
Binding the system 2-6
Building the system 2-6
ICUMRG 1-49
Insufficient access rights for the definition file 1-8
Interactive 1-42
Internal 1-43
Invocation 1-12
System initialization 5-3
Type of 1-42
Examples
Adding a RAM driver B-21
Adding an Intel-device driver B-15
Adding an Intel-supplied controller B-6
Adding unit information B-10, 18
Assembling configuration files B-34
Binding the subsystems B-34
Bootloading a system B-40
Building the system B-34
Changing system memory B-29, C-2, D-2
Configuration C-2, D-2
Deleting a repetitive fixed screen 1-38
Deleting data on a repetitive screen format 1-36
Deleting the System Debugger from the system C-4, D-5
Deleting the UDI from the system D-5
Editing a screen B-6, 7
Formatting the copy routine D-13
Formatting the operating system PROM file C-11, 17, D-12
Generate 2-2, B-32, 44, C-8, D-9
ICUMRG 1-48
Inserting a repetitive-fixed screen 1-42
Inserting data on a repetitive screen format 1-37
Invoking the ICU 1-9, 10, B-4
List command B-44
Loading a system into RAM B-40
Memory adjustments to include the Bootstrap Loader C-3
Memory adjustments to include the iSDM™ monitor C-3
Memory for Free Space Manager Adjustments C-4, D-4
Memory map for 80286-based system A-3, B-28
Programming a 286-based system into PROM devices C-1
Programming a 386/100-based system into PROM devices D-1

Examples (cont.)
Programming a 386/20-based system into PROM devices D-15
Programming PROM devices C-10, D-10
Programming the Bootstrap Loader into PROM devices C-14
Programming the iSDM™ monitor into PROM devices C-13
Programming the operating system into PROM devices C-12, 18, D-13
Removing device drivers C-5
Starting the operating system in ROM C-21
Starting the operating system in ROM from the iSDMa™ monitor C-15
System configuration B-1
Exit command 1-25, 2-1, B-34

F

File locations 1-2
File version numbers 1-12, 44, 46
Files of the ICU 1-2, 5, 6, 2-3, A-1
Finding a screen 1-33, 34
Fixed screen formats 1-30
Formatting the copy routine D-13
Formatting the operating system PROM file C-11, 17, D-12
Formats of screens 1-30

G

General ICU use 1-3
Generate command 1-18, 2-1, B-32, 44, C-8, D-9
Generated files 2-3, A-1
Generating configuration files 2-1, C-8, D-9
Generating the system 2-1, B-31
Getting help 1-15, 30, 33, 34

H

Hardware requirements 1-4
Hardware screen 1-28
Help command 1-15, 30
Help for special editing commands 1-33, 34
How to choose a definition file 1-7

I

I^2ICE™ In-Circuit Emulator 5-6
ICU flowchart 1-39
ICUMRG Utility 1-45
Include files 3-2

Including the Bootstrap Loader C-9
Including the iSDM™ monitor C-9
Initialization routine 3-8
Initializing your system 5-1
Inserting a new line 1-33, 34, 37
Inserting a new screen 1-33, 34, 42
Inserting a repetitive-fixed screen 1-42
Inserting data on a repetitive screen format 1-37
Intel-supplied definition files 1-7
Interactive errors 1-42
Interface libraries 3-5
Internal errors 1-43
Invocation errors 1-12
Invoking the ICU 1-9, 10, 11, B-4
ISDM™ monitor inclusion C-9

L

Language requirements when writing application code 3-2
Level of detail for screens 1-22
List command 1-19, 2-1, B-44
Listing a definition file 1-19
Loading the system into RAM 5-1, B-40
Location of ICU files 1-2
Log-file 1-45
Logical flow of the ICU 1-39

M

Main menu screen 1-14, B-5
Manual overview iii
Memory locations 3-3, 6, B-29
Minimizing memory address size 3-6, B-41

N

Naming ICU-generated files 1-11
Non-resident users 4-1

P

Parts of a screen 1-26
Pre-configuration requirements 1-4
Prefix option 1-11, 2-3
Preparing a RAM-based system 3-1
Preparing application code 3-1

Product overview 1-1
Programming PROM devices C-1, 10, D-10
Programming the Bootstrap Loader into PROM devices C-14
Programming the iSDM™ monitor into PROM devices C-13
Programming the operating system into PROM devices C-18, 12, D-13

Q

Quit command 1-20

R

RAM disk driver 3-3
RAM-based systems 3-1
Reader level iii
Redisplaying the current screen 1-33, 34
Repetitive screen formats 1- 30
Repetitive-fixed screen formats 1-31
Replace command 1-21
Resident user 4-1
Restoring from a file 1-13, 23, 44
Returning to command mode 1-33, C-7, D-9
ROM-based systems 3-7, C-1, D-1

S

Save command 1-19, 25, 2-1
Saving an edited definition file 1-19, 21, 25
Screen editing 1-29
Screen editing commands 1-32, 33
Screen elements 1-26
Screen formats 1-30
Screen names 1-18
Searching for a string within a screen 1-33, 35
Soft-Scope 286® 5-5
Software requirements 1-4
Special editing commands 1-33
Start address of a system 3-8
Starting the operating system in ROM C-21
Starting the operating system in ROM from the iSDM™ monitor C-15
Submit files 1-18, 2-2, 4, B-34, 46, C-8, D-10
Synchronous initialization 5-2
System debugger 5-5

T

Testing the system 5-1, 4

U

Unit information, adding B-10, 18
UPDEF Utility 1-1, 7, 44
Upgrading definition files 1-43
Using the ICU 1-3, 39

V

Version numbers, files 1-12, 44, 46

W

When to use the ICU 1-2
Writing Application code 3-1

intel

EXTENDED iRMX®II PROGRAMMING TECHNIQUES REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

INTRODUCTION

This manual summarizes techniques that will be useful to you as you produce an application system based on the Extended iRMX II Operating System. A typical development process goes through these stages:

- Dividing the application into jobs and tasks
- Writing the code for tasks
- Writing interrupt handlers
- Configuring and starting up the system
- Debugging an application

Use this manual as a reference guide when developing your application system. The techniques described here will help you save time and avoid problems during the development process.

Information has been added to this manual to reflect changes in the use of external declaration INCLUDE files. Use of individual files, hints for efficiency, and tables of the Nucleus, Basic I/O System, Extended I/O System, Human Interface, Application Loader, and Universal Development Interface system calls with their corresponding external declaration file names appear in Chapter 2.

READER LEVEL

This manual assumes that you are familiar with the following:

- the PL/M-286 programming language
- PL/M-286 segmentation models
- iRMX II jobs, tasks, mailboxes, physical files or named files, stream files, type managers and composite objects, system calls, and segments
- the System Debugger

- object module linking
- object libraries
- programming in the iRMX II environment using PL/M-286

MANUAL OVERVIEW

This manual is organized in the following manner:

Chapter 1	This chapter provides information on PL/M-286 segmentation models.
Chapter 2	This chapter describes how to invoke iRMX II system calls from your source code.
Chapter 3	This chapter describes how communication occurs between iRMX II jobs.
Chapter 4	This chapter presents guidelines for stack sizes.
Chapter 5	This chapter describes how to convert iRMX I (IRMX 86) applications to iRMX II applications. This chapter also describes how to improve the performance of your application.
Appendix A	This appendix lists and describes sample iRMX II applications.

CONVENTIONS

The following conventions are used throughout this manual:

- User input appears in one of the following forms:

```
as bolded text within a screen
```

- The term "iRMX II" refers to the Extended iRMX II.3 Operating System.
- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- All numbers, unless otherwise stated, are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

Intel®

CONTENTS

CHAPTER 1 **PAGE**
SELECTING A PL/M-286 SEGMENTATION MODEL

1.1 Introduction 1-1
1.2 What are the Segmentation Models? 1-2
1.3 Restrictions 1-2
1.3.1 Small Model Restrictions 1-3
1.3.2 Compact Model Restrictions 1-3
1.3.3 Medium Model Restrictions 1-3
1.3.4 Using Large Model 1-3
1.4 Choosing a Segmentation Model 1-3

CHAPTER 2 **PAGE**
USING iRMX® II SYSTEM CALLS

2.1 Introduction 2-1
2.2 Coding the System Calls 2-1
2.2.1 Invoking System Calls From PL/M-286 2-1
2.2.2 Invoking System Calls From Assembly Language 2-1
2.2.3 Invoking System Calls From C 2-4
2.3 Including External Declaration Files 2-4
2.3.1 Using the PASCAL-286 Include File 2-6
2.3.2 Using the FORTRAN-286 Include File 2-7
2.4 Binding Your Code to Interface Libraries 2-11
2.5 Bind Sequence 2-12

CHAPTER 3 **PAGE**
COMMUNICATION BETWEEN iRMX® JOBS

3.1 Introduction 3-1
3.2 Passing Data Between Jobs 3-2
3.3 Passing Objects Between Jobs 3-3
3.3.1 Passing Objects Through Object Directories 3-4
3.3.2 Passing Objects Through Mailboxes 3-6
3.3.3 Passing Parameter Objects 3-6
3.3.4 Restrictions 3-6
3.3.5 Comparison of Object-Passing Techniques 3-7

CHAPTER 4 **PAGE**
GUIDELINES FOR STACK SIZES

4.1 Introduction 4-1
4.2 Stack Size Limitation for Interrupt Handlers 4-1
4.3 Stack Guidelines for Creating Tasks and Jobs 4-2
4.4 Stack Guidelines for Tasks to be Loaded or Invoked 4-2

CHAPTER 4 (continued) **PAGE**

4.5 Arithmetic Technique 4-3
4.5.1 Stack Requirements for Interrupts 4-3
4.5.2 Stack Requirements for System Calls 4-3
4.5.3 Computing Stack Size 4-4
4.6 Empirical Technique 4-4

CHAPTER 5 **PAGE**
CONVERTING iRMX® I APPLICATIONS AND IMPROVING PERFORMANCE

5.1 Introduction 5-1
5.2 Effects of Segmentation 5-1
5.2.1 Causes of Program Slowdown 5-1
5.2.2 Avoiding Slowdown 5-2
5.3 PL/M-286 Based Variables 5-2
5.4 Optimizing Nucleus Performance 5-3
5.5 Optimizing Sequential I/O 5-3
5.6 Mailbox Tuning 5-3
5.7 Recycling Buffers 5-3
5.8 Conserving Limited Resources 5-4

APPENDIX A **PAGE**
EXAMPLE PROGRAMS

A.1 Introduction A-1
A.2 Example 1 - System Programming Concepts A-1
A.2.1 Program Source Code A-2
A.2.2 Concepts A-2
A.2.2.1 In-Line Exception Processing A-3
A.2.2.2 Use of Literal Files A-5
A.2.2.3 Getting and Setting Terminal Attributes A-7
A.2.2.4 Creating Tasks A-9
A.2.2.5 Cataloging Objects A-10
A.2.2.6 Using a Response Pointer During Inter-Task Communication A-12
A.2.2.7 Using Buffer Pools A-15
A.2.2.8 Methods of Screen Input/Output A-20
A.2.2.9 Simultaneous Input/Output A-22
A.2.3 Compiling and Binding the Code A-23
A.2.4 Running the Example A-24
A.3 Example 2 - Task Communication A-26
A.3.1 Program Source Code A-26
A.3.2 Include Files A-36
A.3.3 Compiling and Binding the Code A-37
A.3.4 Running the Example A-38
A.4 Example 3 - Control-C Handler A-38
A.4.1 Source Code A-38
A.4.2 Compiling and Binding the Code A-41
A.4.3 Running the Example A-41

APPENDIX A (continued) **PAGE**

A.5 Hardware Requirements for Remaining Examples A-42
A.6 Example 4 - Interrupt Task A-43
A.6.1 Source Code A-43
A.6.2 Compiling and Binding the Code A-48
A.6.3 Running the Example A-48
A.7 Example 5 - First-Level Job A-48
A.7.1 Source Code A-49
A.7.2 Compiling and Binding the Code A-80
A.7.3 Configuring the First-Level Job A-81

TABLE **PAGE**

2-1 iRMX® II External Declaration INCLUDE Files 2-5
2-2 Interface Libraries and iRMX® II Layers 2-11
4-1 Stack Requirements for Interrupts and System Calls 4-4
A-1 Literal Files Helpful With Nucleus System Calls A-5
A-2 Literal Files Helpful With BIOS System Calls A-6
A-3 Literal Files Helpful With EIOS System Calls A-6
A-4 Literal Files Helpful With Human Interface System Calls A-6

FIGURES

FIGURE		PAGE
A-1	Example PL/M-286 Application (INIT)	A-28
A-2	Example PL/M-286 Application (ALPHONSE)	A-31
A-3	Example PL/M-286 Application (GASTON)	A-33
A-4	CONTROL-C Handler Example	A-39
A-5	Interrupt Task Example	A-44
A-6	First-Level Job Initialization Task (INITTSK.P28)	A-52
A-7	Terminal Job (TERM.P28)	A-55
A-8	Interrupt Handler and Task (SBX350.P28)	A-63
A-9	Process Task (PROCIO.P28)	A-67
A-10	Utility Procedures (UTILS.P28)	A-76
A-11	User Jobs Screen	A-83

CHAPTER 1
SELECTING A PL/M-286 SEGMENTATION MODEL

1.1 INTRODUCTION

Read this chapter only if you will be programming iRMX II tasks using PL/M-286. You should already be familiar with the following concepts:

- The PL/M-286 programming language
- PL/M-286 segmentation models
- iRMX II jobs, tasks, and segments

When you invoke the PL/M-286 compiler, you must specify (either explicitly or by default) the segmentation model (SMALL, COMPACT, MEDIUM, or LARGE) that your program will use. The segmentation model affects the amount of memory required to store your application's object code and the performance of the application.

With operating systems that run on 8086 processors (or on 80286 and 80386 processors in real address mode), choosing the appropriate segmentation model is important for reducing the amount of memory that an application uses. Memory use is important because the 8086 processor is limited to 1 megabyte of memory address space. The operating system, on the other hand, uses the 80286 and 80386 processors in protected virtual address mode (PVAM). In PVAM, the processor can access the full memory address range that is available. Thus, program size becomes less important.

However, loading a segment register in PVAM takes longer than loading other kinds of instructions. If you can minimize the number of times the processor must switch segments, application performance will improve. Because the segmentation model determines how code and data are stored, choosing the proper model minimizes the number of segment switches, and increases performance.

The following sections explain which segmentation model will attain the highest performance, while still satisfying system requirements.

1.2 WHAT ARE THE SEGMENTATION MODELS?

The four segmentation models supported by PL/M-286 are SMALL, COMPACT, MEDIUM, and LARGE. The *PL/M-286 User's Guide for iRMX 286 Systems* describes each of these models in detail. This section gives a brief overview of each.

SMALL	Code sections from all linked modules are placed in the same code segment, which is addressed by CS. Data and stack sections are placed in the same data segment, which is addressed by both DS and SS.
COMPACT	Code sections from all linked modules are placed in the same code segment, which is addressed by CS. Data sections are placed into a single data segment, which is addressed by DS. Likewise, stack sections are placed into a stack segment, which is addressed by SS.
MEDIUM	The code section from each compiled module is placed in its own code segment, enabling the total amount of code to be more than 64K bytes. Data and stack sections from all linked modules are placed into a single data segment, which is addressed by both DS and SS. During program execution, the CS register is updated whenever a PUBLIC or EXTERNAL procedure is activated.
LARGE	Code and data sections from each compiled module are placed into their own code and data segments, enabling the total amount of code and data to be more than 64K bytes. Stack sections are placed into a single stack segment, which is addressed by SS. In LARGE model, code and data segments are paired. During program execution, both the CS and DS are updated whenever a PUBLIC or EXTERNAL procedure is activated.

Specifying the ROM or RAM compiler controls determines whether the constants you define in your programs are placed in the code or data areas. This provides additional control on the size of those segments.

1.3 RESTRICTIONS

The fewer times your application must load the segment registers, the better it performs. To improve performance, choose the segmentation model that uses the fewest segments but still supports the required amount of code or data. This practice means starting with the SMALL model and working up to the LARGE model, using the first model that can handle the amount of code or data in your application. However, some models place restrictions on iRMX II operations.

1.3.1 Small Model Restrictions

When you compile programs using the PL/M-286 SMALL control, all POINTER values are 16 bits long. This introduces some restrictions, including inability to address the contents of an iRMX II segment received from another job. Because of these restrictions, the only applications that can use SMALL model are those that invoke UDI system calls only. Applications that invoke other iRMX II system calls cannot use the SMALL segmentation model.

1.3.2 Compact Model Restrictions

You cannot compile exception handlers in the COMPACT model and link them with other COMPACT procedures, because the operating system always makes a far call to an exception handler. To include exception handlers, compile them using the LARGE size control.

1.3.3 Medium Model Restrictions

When using PL/M-286 MEDIUM model, you lose the option of having the Operating System dynamically allocate stacks for tasks that are created dynamically. Anticipate each task's stack requirements, and explicitly reserve memory for each stack during configuration.

1.3.4 Using Large Model

There are no restrictions with the PL/M-286 LARGE model. If your application is too large for the other models, or you wish to avoid restrictions, use the LARGE model.

1.4 CHOOSING A SEGMENTATION MODEL

For best performance, use the following guidelines when choosing a segmentation model:

- If your code and data can each fit into a 64K-byte segment, use the COMPACT model.
- If your application is too large for COMPACT, consider using COMPACT subsystems. This enables you to set up your application in pieces (subsystems), each of which adheres to the COMPACT model. With COMPACT subsystems, segment registers are changed only when you call procedures or access variables that reside in one of the other subsystems. However, creating COMPACT subsystems requires more technical knowledge than the other alternatives and it requires changes to the source code. Refer to the *PL/M-286 User's Guide for iRMX 286 Systems* for more information.

- If you decide not to use COMPACT subsystems, MEDIUM offers the next best performance. MEDIUM, however, requires that all data and stack fit into a single 64K segment.
- If all else fails, use LARGE model. There are no size or iRMX II restrictions with LARGE, but this model results in the largest number of segment register switches.

To determine whether your application can fit into the COMPACT model, try compiling and binding it under the COMPACT model. If the application is too large for COMPACT, BND286 will return an error message. At that point, you can decide whether to use MEDIUM, LARGE, or recode your application for COMPACT subsystems.

If your application will be loaded into RAM, you may be able to use the ROM or RAM controls to adjust segment sizes so that your application fits into the COMPACT or MEDIUM models. These controls specify where your program's constants will reside. For example, if your application's data is slightly larger than 64K bytes, specifying the ROM control (which places the constants in the code segment) might allow the remaining data to fit in a 64K segment. This could make your code eligible for the COMPACT or MEDIUM models.

CHAPTER 2
USING iRMX® II SYSTEM CALLS

2.1 INTRODUCTION

Read this chapter if you write programs that use iRMX II system calls. You should already be familiar with the following concepts:

- System calls
- Object module linking
- Object libraries
- PL/M-286 segmentation models

This chapter explains how to include iRMX II system calls in your programs, and how to bind your code with the necessary iRMX II libraries.

2.2 CODING THE SYSTEM CALLS

The first step in invoking iRMX II system calls is placing source statements in your code. The following sections discuss how to place system calls in your PL/M-286 and ASM286 source code.

2.2.1 Invoking System Calls From PL/M-286

The iRMX II system call reference manuals use the PL/M-286 syntax when listing the iRMX II system calls. When writing code, use the syntax listed in the system call reference manuals.

2.2.2 Invoking System Calls From Assembly Language

Programs communicate with the operating system calling interface procedures designed for use with programs written in PL/M-286. To invoke system calls from assembly language programs, the assembly language programs must obey the procedure-calling protocol used by PL/M-286. For example, if your ASM286 program uses the SEND$MESSAGE system call, then you must call the rq$send$message interface procedure from your assembly language code.

The technique for calling PL/M-286 procedures from assembly language is described in the *ASM286 Macro Assembler Operating Instructions for iRMX 286 Systems*. This section presents an overview of the technique.

In general, to call a PL/M-286 procedure, first push all the parameters onto the stack and then call the procedure. Push the parameters in the order they are listed in the system call reference manuals; that is, starting with the leftmost parameter. Long pointers (complete addresses consisting of a selector and an offset) should be pushed as two words: the selector first, then the offset.

The CALL instruction also places the return address of your calling procedure onto the stack. This enables control to return to your program after the system call completes.

Some system calls return values. In assembly language, the returned values are available in registers, as follows:

Type	Register
BYTE	AL
WORD	AX
DWORD	DX:AX
INTEGER	AX
POINTER	ES:BX
SELECTOR	AX

When writing assembly language routines that call PL/M-286 interface procedures, you must adhere to a segmentation model (COMPACT, MEDIUM, or LARGE) because conventions for making calls depend on the segmentation model.

If your application is written entirely in assembly language, you can arbitrarily select an interface library (COMPACT or LARGE) based on whether your application makes near or far calls. Size and performance advantages can be gained by using the COMPACT interface procedures, because their procedure calls are all NEAR. The LARGE interface, which has procedures that require FAR procedure calls, is advantageous only if your application code is larger than 64K bytes.

However, if some of your application code is written in PL/M-286, your assembly language code should use the same interface procedures as those used by your PL/M-286 code.

The following example shows how to call iRMX II system calls from assembly language. The example assumes that the COMPACT segmentation model is used.

```
DATA segment RW PUBLIC

seg_tok  DW ?
excep    DW ?

DATA ENDS

CODE segment ER PUBLIC

extrn rqcreatesegment:  near

my_prog PROC near
;
; Get addressability to parameters
;
push  bp
mov   bp, sp
;
; Save caller's DS and obtain local DS
;
push  ds
mov   ds, data
    .
    .
  Typical ASM statements
    .
    .
;
; seg_tok = rq$create$segment (400H, @excep);
;
push  400H
push  ds
push  offset excep
call  rqcreatesegment
mov   seg_tok, ax
```

```
;
; IF except <> E$OK THEN GOTO error;
;
cmp  excep, 0
jnz error
    .
    .
  Typical ASM statements
    .
    .
my_prog ENDP
CODE    ENDS
END
```

2.2.3 Invoking System Calls From C

Programs written in C can easily directly access iRMX II system calls by defining them as alien procedures.

For example, the following lines define the CREATE$SEGMENT and DELETE$SEGMENT system calls as alien procedures.

```
alien unsigned short rqcreatesegment()
alien rqdeletesegment()
```

If you invoke system calls from C, you must pass parameters of the same type expected by the system calls. The C compiler does no type checking of alien procedures. Rather, it assumes the parameter types are those that are used in the actual call. In particular, 32-bit constants should be explicitly marked by using the L suffix.

2.3 INCLUDING EXTERNAL DECLARATION FILES

When you call a procedure that is not defined in your current program module (a separately compiled portion of your program), you must declare that procedure to be external. Doing so enables the binder to satisfy the references to that procedure when it links your program modules together. This enables a program in one module to call a procedure in another module.

Invoking iRMX II system calls is just like calling any PL/M-286 procedure. Because you don't define the system calls in your programs, they must be external procedures. Therefore, you must include external declarations for each system call you invoke.

Intel has made it easy for you to place external declarations for the system calls in your programs. Supplied on the iRMX II release diskettes are INCLUDE files, which reside permanently in one location and provide the PL/M-286 external procedure declarations for all iRMX II and UDI system calls. An INCLUDE file eliminates duplication of statements in your source code modules. The declarations are written once, placed in an INCLUDE file, and then used in lieu of repeating the actual declaration in each module.

For example, to use the INCLUDE file NUCLUS.EXT, place the following statement at the beginning of your PL/M-286 source code. This statement declares all the Nucleus system calls to be external.

```
$INCLUDE(/RMX286/INC/NUCLUS.EXT)
```

Table 2-1 lists the external declaration INCLUDE files supplied with the operating system. After installation of the operating system, these files are available in the /RMX286/INC directory.

Table 2-1. iRMX® II External Declaration INCLUDE Files

INCLUDE File	Language	Description
RMXPAS.EXT	PASCAL-286	External declarations for all system calls. Refer to the "Using the PASCAL INCLUDE File" section for more information.
RMXFTN.EXT	FORTRAN-286	External declarations for all system calls that return a value. Refer to the "Using the FORTRAN INCLUDE File" section for more information.
RMXPLM.EXT	PL/M-286	External declarations for all iRMX II system calls.
NUCLUS.EXT	PL/M-286	External declarations for Nucleus system calls.
BIOS.EXT	PL/M-286	External declarations for Basic I/O System calls.
EIOS.EXT	PL/M-286	External declarations for Extended I/O System calls.
LOADER.EXT	PL/M-286	External declarations for Application Loader system calls.
HI.EXT	PL/M-286	External declarations for Human Interface system calls.
UDI.EXT	PL/M-286	External declarations for UDI system calls.

Because each INCLUDE file contains external declarations for many system calls (either all iRMX II system calls or all system calls in a particular subsystem), including a particular file will probably result in external declarations for several system calls your program does not invoke. Although having extra external declarations poses no problems for the compilers and causes no error conditions, you can improve compilation speed by copying just the external declarations your program needs into a separate INCLUDE file and specifying that file in the INCLUDE statement.

2.3.1 Using the PASCAL-286 Include File

When using PASCAL-286 to write an application, you must include the RMXPAS.EXT INCLUDE file with your PASCAL-286 source program. To do this, enter this statement in your source program:

```
$INCLUDE(/RMX286/INC/RMXPAS.EXT)
```

This statement inserts the contents of RMXPAS.EXT into your PASCAL-286 program. This file contains external declarations for all the iRMX II system calls. However, it uses names that are slightly different from the system call names shown in the reference manuals. Because PASCAL does not allow dollar signs ($) in procedure names, the system calls are declared without the $ characters. To make the names recognizable, the INCLUDE file mixes uppercase and lowercase letters in the names. You can use this convention in your programs also.

Because of data-type checking that the PASCAL-286 compiler performs, the INCLUDE file may require editing before your program will compile correctly. Four Nucleus and two EIOS system calls are affected:

Nucleus System Calls	EIOS System Calls
RQCreateJob	RQCreateIOJob
RQECreateJob	RQECreateIOJob
RQCreateTask	
RQSignalException	

These six system calls enable you to specify an absolute value (0) or a POINTER (such as @stacktop) as the STACKPTR parameter. No single PASCAL-286 data type can be used to declare both types of values.

RMXPAS.EXT contains a declaration for specifying pointers. This declaration enables you to specify a stack location by supplying a pointer to the stack.

You may want to use the value 0, which indicates that the operating system is to provide the application's stack location, for the STACKPTR parameter.

To let the operating system provide the stack, edit RMXPAS.EXT as shown in the following paragraph and use comment notation around the form of the declaration that is no longer needed. The following example changes the declaration of the STACKPTR parameter in the Nucleus call RQ$CREATE$TASK, but the form applies to all five system calls that may need editing. Note that only one STACKPTR declaration can be "active."

Change

```
FUNCTION RQCREATETASK(
                      PRIORITY        : BYTE;
                VAR   STARTADDRESS    : BYTES;
                      DATASEG         : WORD;
                VAR   STACKPTR        : BYTES /STACKADDR/;
                      {STACKPTR       : LONGINT;}
                      STACKSIZE       : WORD;
                      TASKFLAGS       : WORD;
                VAR   EXCEPTR         : BYTES);
```

to read

```
FUNCTION RQCREATETASK(
                      PRIORITY        : BYTE;
                VAR   STARTADDRESS    : BYTES;
                      DATASEG         : WORD;
               {VAR   STACKPTR        : BYTES /STACKADDR/;}
                      STACKPTR        : LONGINT;
                      STACKSIZE       : WORD;
                      TASKFLAGS       : WORD;
                VAR   EXCEPTR         : BYTES);
```

Note that you can change any or all five declarations, depending on the requirements of your own PASCAL-286 program.

2.3.2 Using the FORTRAN-286 Include File

When you use FORTRAN-286 to write an application, you must include the RMXFTN.EXT INCLUDE file with your FORTRAN-286 source programs. To do this, place the following statement in your source code:

```
$INCLUDE(/RMX286/INC/RMXFTN.EXT)
```

This statement inserts the contents of RMXFTN.EXT into your FORTRAN-286 program. Only those system calls that return values need to be declared, so not all system calls are contained in the file RMXFTN.EXT.

When invoking iRMX II system calls from FORTRAN-286 programs, remember the following techniques:

- Use the %VAL pseudo function when passing any parameter that is not a pointer. FORTRAN passes parameters by reference; the %VAL pseudo function forces the parameter to be passed by value. When passing parameters, match the type of the formal parameter. That is, variables of WORD or BYTE types should be passed as words on the stack; DWORDs should be passed as two words. You must do this to assign parameters to a variable and then pass the variable using %VAL. This method is mandatory for expressions (even built-ins such as INT2), because expressions cannot be passed using %VAL.
- Because FORTRAN does not recognize unsigned entities, you may have difficulty referring to byte values greater than or equal to 128, word values greater than or equal to 32768, and double word values greater than or equal to 2**31. For example, the GET$PRIORITY system call returns an unsigned byte value that indicates the priority. To deal with this number in FORTRAN, either check the value as a negative number, or assign the value to a word and mask the high byte to zero. These examples show how to deal with byte and word values.

  Dealing with byte values greater than or equal to 128

```
      integer*2 task_tok, status, pri_word
      integer*1 pri
          .
          .
          .
      pri=RqGetPriority(%val(task_tok),status)
          .
          .
          .
C     convert the signed byte to an unsigned word
      pri_word = pri.and.#0ff
```

Dealing with word values greater than or equal to 32768

```
      integer*2 seg_tok, status, size
      integer*4 size_dword
            .
            .
            .
      size=RqGetSize(%val(seg_tok),status)
            .
            .
            .
C     convert the signed word to an unsigned dword
      size_dword = size.and.#0ffff
```

- FORTRAN strings are not iRMX II strings and are not useful for any iRMX II system calls. A FORTRAN string should be converted into an array, with the first byte set to the length of the string. The following example illustrates this technique.

```
      SUBROUTINE RMXSTR(FILE_NAME,FILE_ARRAY)
      CHARACTER*(*) FILE_NAME
      INTEGER*1 FILE_ARRAY(*)
      INTEGER*2 LN,I

      LN=LEN(FILE_NAME)

C     move string to array
      FILE_ARRAY(1)=LN
      DO 100,I=1,LN
          FILE_ARRAY(I+1)=ICHAR(FILE_NAME(I:I))
100   CONTINUE

C     suppress trailing blanks, fill leading length byte
      DO 200,I=1,LN
          IF (CHAR(FILE_ARRAY(LN+2-I)).NE.' ') THEN
              FILE_ARRAY(I)=LN+1-I
              RETURN
          ENDIF
200   CONTINUE
      END
          .
          .
          .
      character*40 file_name
      integer*1 file_array(41),co(5)
          .
          .
          .
      call rmxstr(':co:',co)
          .
          .
          .
      file_name='/rmx286/lib/source/a.x'
          .
          .
          .
      call rmxstr(file_name,file_array)
          .
          .
          .
      end
```

2.4 BINDING YOUR CODE TO INTERFACE LIBRARIES

After you have written your programs and inserted INCLUDE statements for the necessary system calls, you must compile the code and bind it to the appropriate iRMX II interface library.

Interface libraries, supplied with the iRMX II product, provide a standard and simple interface to the system calls. The interface libraries contain procedures that correspond to iRMX II system calls. These procedures have the same names and use the same parameters as the system calls. To invoke a system call, simply declare the system calls as external procedures (as described in the previous section) and call the corresponding interface procedure just as you would a PL/M-286 procedure. The interface procedure performs more complicated operations to invoke the actual system call. For example, iRMX I interface procedures use software interrupts to invoke system calls. Their iRMX II counterparts make calls to call gates when accessing system calls.

After compiling the program code, you must satisfy the external references to the system calls by using BND286, which binds the compiled code to the appropriate interface libraries. There are several interface libraries from which to choose. Which interface library must be bound to your program depends on the system calls and the segmentation model used. Table 2-2 lists the interface libraries.

As Table 2-2 shows, for each segmentation model, except SMALL, there is one interface library for UDI and one interface library for all the other layers. The SMALL model is supported only for UDI. If your code includes only UDI system calls (or if it uses the I/O support provided by the language), bind your program only to the appropriate UDI library. If your code doesn't invoke UDI system calls, or you don't plan to include the language's I/O support, bind the code just to the appropriate RMXxxx library. If your code invokes both UDI and other iRMX II system calls, bind the code to both of the libraries for the segmentation model you chose. In this last instance, when you specify the BND286 command to bind your code, the UDI interface library must precede the RMXxxx library in the list of modules to be bound.

Table 2-2. Interface Libraries and iRMX® II Layers

	SMALL	COMPACT	LARGE OR MEDIUM
All layers except UDI		RMXIFC.LIB	RMXIFL.LIB
UDI	UDIIFS.LIB	UDIIFC.LIB	UDIIFL.LIB

2.5 BIND SEQUENCE

The previous section described which interface libraries you should bind to your program's object code. This section discusses the entire BND286 command that should be used to bind a program in preparation for execution under the Operating System.

The following example shows the bind sequence for a PL/M-286 program that uses the COMPACT segmentation model. The program consists of three object modules, MODA.OBJ, MODB.OBJ, and MODC.OBJ. The program invokes UDI and other iRMX II system calls. After binding, the resulting executable module will be placed in a file called FINALSYS. It is assumed that FINALSYS will be invoked from the Human Interface; that is, by typing its name at the Human Interface prompt.

```
BND286  &
MODA.OBJ, MODB.OBJ, MODC.OBJ,    &
:LANG:PLM286.LIB,    &
/RMX286/LIB/UDIIFC.LIB, &
/RMX286/LIB/RMXIFC.LIB  &
OBJECT(FINALSYS)     &
SEGSIZE(STACK(+1750))    &
RCONFIGURE(DYNAMICMEM(5000H, 0FFFFH))
```

In this BND286 statement, the three object files (MODA.OBJ, MODB.OBJ, and MODC.OBJ) are bound together with three libraries: PLM286.LIB, UDIIFC.LIB and RMXIFC.LIB. The library called PLM286.LIB is the standard PLM-286 library distributed with the compiler. It satisfies compiler-generated externals, such as those that occur when you call built-in functions (DWORD arithmetic is one example). PLM286.LIB should always be the first library that you bind to your PLM-286 object code.

The second and third libraries (UDIIFC.LIB and RMXIFC.LIB) are the iRMX II interface libraries used with the COMPACT segmentation model. These interface libraries satisfy the external references generated when your programs invoke UDI and iRMX II system calls. UDIIFC.LIB should always precede RMXIFC.LIB in the bind sequence.

The OBJECT control specifies the name of the executable file to be generated by BND286. In this case, the file is called FINALSYS.

The SEGSIZE(STACK(+1750)) control specifies that an additional 1750 decimal bytes of stack beyond that required by the program alone should be reserved for this application. These extra bytes are required for invoking system calls. Refer to Chapter 7 for guidelines on selecting stack sizes.

The RCONFIGURE(DYNAMICMEM(5000H, 0FFFFH)) control directs BND286 to produce a single-task loadable (STL) module, to assign a minimum of 5000H bytes of dynamic memory to the module, and to limit the amount of memory it can borrow from its parent to 0FFFFH bytes. All iRMX II tasks to be loaded by the Application Loader must be STL modules (and programs invoked at the Human Interface level are loaded by the Application Loader). Therefore, your BND286 command should always include the RCONFIGURE control.

CHAPTER 3
COMMUNICATION BETWEEN iRMX® II JOBS

3.1 INTRODUCTION

Read this chapter if you want to pass information from one iRMX II job to another. You should already be familiar with the following concepts:

- iRMX II jobs, including the root job and object directories
- iRMX II tasks
- iRMX II segments
- iRMX II mailboxes
- iRMX II physical files or named files
- iRMX II stream files
- iRMX II type managers and composite objects

In multiprogramming systems, where each of several applications is implemented as a distinct iRMX II job, information must occasionally pass from one job to another. This chapter describes several techniques you can use to accomplish this.

The techniques are divided into two groups: passing data between jobs, and passing iRMX II objects.

NOTE

Many of the techniques in this chapter involve sending tokens or pointers to other jobs. If the sending job is deleted, these entities become invalid. If the receiving job uses the entities later, the results are unpredictable (usually a general protection error). It is the programmer's responsibility to avoid these situations.

3.2 PASSING DATA BETWEEN JOBS

Data can be sent from one job to another in several ways:

1. You can use the SEND$DATA and RECEIVE$DATA system calls to exchange the information. SEND$DATA and RECEIVE$DATA use a mailbox to exchange up to 128 bytes of information. To use this method, the task that creates the mailbox must catalog it in the root object directory, so that the other task can access it. This procedure is described later in this chapter.

   The advantages of this technique are

   - Because this technique requires only the Nucleus, you can use it in systems that do not use other iRMX II layers.
   - Although the Operating system copies the information from one place to another, it copies the information very quickly.
   - It is the quickest method of passing small amounts of data.
   - It can be used to exchange larger amounts of data by passing the pointer to that data as data itself in the SEND$DATA system call. The receiving job can then use the pointer to refer to the relevant data. This method of exchanging data is faster than using SEND$MESSAGE because it does not require the overhead of creating a segment.

   The disadvantages of this technique are

   - Only 128 bytes of information can be exchanged with the SEND$DATA and RECEIVE$DATA system calls.

2. You can create an iRMX II 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

   - Because this technique requires only the Nucleus, you can use it in systems that do not use other iRMX II layers.
   - The Operating system does not copy the information from one place to another.

   The disadvantages of this technique are

   - The segment occupies 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 or from it.

3. You can use an iRMX II stream file.

   The advantages of this technique are

   - You can use I/O System calls to pass information directly between tasks, without slowing down the transfer by temporarily placing the data on a secondary storage device.
   - This technique can easily be changed to Technique 4 (below).

   The disadvantages of this technique are

   - You must configure one or both I/O systems into your application system.
   - There is a 4K byte limit on the amount of data you can pass via a stream file.
   - Writing data to and reading data from a stream file is much slower than any of the previous methods.

4. As a last resort, you can use the Basic I/O System, the Extended I/O System, or the Universal Development Interface 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 3 (above).

   The disadvantages of this technique are

   - You must incorporate one or both I/O systems (and possibly UDI) into your application system.
   - Device I/O is slower than reading and writing to a stream file, making this the slowest of the four methods.

3.3 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. In the following discussions you will see how to pass objects by using object directories, mailboxes, and parameter objects.

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), Job B cannot successfully use the object to perform I/O. Instead, Job B must create another connection to the same file. It does this by invoking the Basic I/O System's A$ATTACH$FILE system call, using the connection obtained from Job A as the prefix parameter. No subpath parameter needs to be specified. A$ATTACH$FILE returns a new connection to the same file, which Job B can use to perform I/O. This restriction about connections is discussed in the *Extended iRMX II Basic I/O System User's Guide* and in the *iRMX II Extended I/O System User's Guide*.

3.3.1 Passing Objects Through Object Directories

Consider a hypothetical system in which tasks in separate jobs must communicate with each other. 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 were 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 solution 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.

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, 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.

You should be aware of several aspects of this technique:

- In the example, the object directory technique was used to pass a semaphore. You can use the same technique to pass any type of iRMX II object.
- The semaphore was passed via the object directory of the root job. The root job's object directory is unique because 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 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. To avoid this, use an object directory other than the root object directory for different sets of jobs. For example, have one of the jobs catalog its token in the root job's object directory under a previously set name. Any other jobs can then look up the token of the cataloged job in the root job's directory, 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 runs first.

3.3.2 Passing Objects Through Mailboxes

You can also send objects from one job to another by using a mailbox. This is a two-step process; the two jobs using the mailbox must first use the object directory technique to obtain mutual access to the mailbox, and then use the mailbox to pass additional objects.

There are two ways to transmit information via a mailbox. If the mailbox is set up to transfer data, the tasks can use SEND$DATA and RECEIVE$DATA to send and receive 128-byte data. With these calls, tasks can transfer strings, arrays, and even tokens for objects, as long as the information is 128 bytes or less.

If the mailbox is set up to transfer objects, the tasks can use SEND$MESSAGE and RECEIVE$MESSAGE to send and receive iRMX II objects. With these calls, only the object tokens are sent and received.

Whenever two jobs send data via mailboxes, the sending and receiving task must perform handshaking to ensure that they are finished using the segment containing the data. If the segment is deleted and the receiving task attempts to access the now nonexistent segment, a general protection error occurs.

3.3.3 Passing Parameter Objects

One of the parameters of the CREATE$JOB system call is a parameter object. This parameter allows a task in the parent job to pass an object to the newly created job. Once the tasks in the new job begin running, they can obtain a token for the parameter object by calling GET$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 needed by Job 2. Task 1 can pass the array to Job 2 by putting the array into an iRMX II 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 example, the parameter object is a segment. However, you can use this technique to pass any kind of iRMX II object.

3.3.4 Restrictions

As mentioned earlier, there is a restriction concerning the passing of connection objects between jobs. When a file connection created in one job is passed to a second job, the second job cannot use the object to perform I/O. Instead, the second job must create another connection to the same file. This restriction is discussed in the *Extended iRMX II Basic I/O System User's Guide* and in the *iRMX Extended I/O System User's Guide*.

3.3.5 Comparison of Object-Passing Techniques

Consider these guidelines 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 two jobs frequently pass objects between each other, the mailbox technique is the fastest and easiest method.
- If you need to pass more than one object at a time, use any of the following techniques:
  - -- If the tokens fit in a 128-byte array, create a mailbox and use SEND$DATA and RECEIVE$DATA to transfer the array.
  - -- Assign an order to the objects and send them one by one to a mailbox (using either SEND$DATA or SEND$MESSAGE) where the receiving job can pick them up in order.
  - -- Give each object 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 4
GUIDELINES FOR STACK SIZES

4.1 INTRODUCTION

This chapter is for three kinds of readers:

- readers who write tasks that create iRMX II jobs or tasks
- readers who write interrupt handlers
- readers who write tasks to be loaded by the Application Loader or tasks to be invoked by the Human Interface

You should already be familiar with the iRMX II System Debugger, and you should know which system calls are provided by the various layers of the Operating System. You also should know the difference between maskable and nonmaskable interrupts.

This chapter will help you compute the amount of stack you must specify for a system call that creates a job or task. If you are writing an interrupt handler, this chapter tells you about stack size limitations that you must adhere to. If you are writing a task to be loaded by the Application Loader or invoked by the Human Interface, this chapter tells you how much stack to reserve during the binding process.

4.2 STACK SIZE LIMITATION FOR INTERRUPT HANDLERS

Many Operating system tasks 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 operating system assumes that all interrupt handlers require no more than 128 (decimal) bytes of stack, even if a task is interrupted by both a maskable and a nonmaskable interrupt. If you fail to adhere to this limitation when writing an interrupt handler, you risk stack overflow in your system.

To stay within the 128 (decimal) byte limitation, restrict the number of local variables that the interrupt handler stores on the stack. For interrupt handlers serving maskable interrupts, you can use as many as 20 (decimal) bytes of stack for local variables. For handlers serving nonmaskable interrupts, 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 *Extended iRMX II Nucleus User's Guide.*

4.3 STACK GUIDELINES FOR CREATING TASKS AND JOBS

When you create a task by invoking a system call, you must specify the size of the task's stack. And, since every new job has an initial task created simultaneously with the job, you must also designate a stack size when you create a job.

Be careful when specifying a task's stack size. Specifying a stack size that is too small, could cause the task to overflow its stack. If the stack overflows, the hardware will detect the error and cause the Nucleus to invoke an exception handler, which could delete the offending task or activate the iSDM monitor. Specifying a stack size that is too large wastes memory. Ideally, you should specify a stack size that is only slightly larger (500-1000 bytes) than what is actually required.

This chapter illustrates two techniques for estimating a task's stack size: arithmetic and empirical. For best results, start with the arithmetic technique and then use the empirical technique to tune your original estimate.

If your programs are recursive, do not rely solely on either of these techniques. Stack usage in recursive routines varies because of run-time events, and should be computed carefully.

To minimize problems, pad the results of your computations by 500 to 1000 bytes to allow for situations that you might not have experienced in your tests.

4.4 STACK GUIDELINES FOR TASKS TO BE LOADED OR INVOKED

If you are creating a task 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 bind process. The following techniques will help you estimate stack size requirements.

4.5 ARITHMETIC TECHNIQUE

This technique slightly overestimates a task's stack size. After you use this technique to obtain an estimate, use the empirical technique described later in this chapter to refine the estimate.

The arithmetic technique is based on these factors, which affect a task's stack:

- Interrupts
- iRMX II 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)

Estimate the size of a task's stack by adding the amount of memory required to accommodate these factors. The following sections explain how to compute these values.

4.5.1 Stack Requirements for Interrupts

When an interrupt occurs while a task is running, the interrupt handler uses the task's stack while it services the interrupt. Consequently, you must ensure that a task's stack is large enough to accommodate the needs of two interrupt handlers: one for maskable interrupts and one for nonmaskable interrupts. Refer to the earlier section "Stack Size Limitation for Interrupt Handlers" for more information.

4.5.2 Stack Requirements for System Calls

When a task invokes an iRMX II system call, the processing associated with the call uses some of the task's stack. The amount of stack required depends on which system calls you use.

Table 4-1 shows how many bytes of stack a task requires to support the system calls of each layer. Included in these figures are the 128 bytes required by the interrupt handlers. To find out how much stack you must allocate for system calls, determine which layers of the operating system your task uses. Scan Table 4-1 to find which of those layers requires the most stack. By allocating enough stack to satisfy the requirements of the most demanding layer, you can satisfy the requirements of all system calls used by your task.

Table 4-1. Stack Requirements for Interrupts and System Calls

Layer	Bytes (Decimal)
UDI	1750
Human Interface	1500
Application Loader	700
Extended I/O System	550
Basic I/O System	350
Nucleus	250

4.5.3 Computing Stack Size

To compute stack size, add the following numbers:

- The number of bytes required for interrupts and system calls, according to the most demanding layer you intend to use.
- The amount of stack required by the task's code. This figure can be determined by looking at the information about the STACK segment in the .MP1 map file that BND286 produces when it binds your application code. This stack usage is the result of calling local procedures and using the stack for local variables when your code is reentrant.

This sum is a conservative, but reasonable, estimate of a task's stack requirements. For more accuracy, use the sum as a starting point for the empirical fine tuning described below.

4.6 EMPIRICAL TECHNIQUE

This technique starts with an overly large stack and uses the iSDM Monitor to determine how much of the stack is unused. Once you have found out how much stack is unused, you can modify your task-creation and job-creation system calls to create smaller stacks.

To use this technique, change your program code to break to the iSDM monitor at the beginning and at the end of the program. When coding in PL/M-286, use the CAUSE$INTERRUPT(3) statement to break to the monitor (INT3 in assembly language). If your application is loaded by the Human Interface (that is, invoked as a command), use the DEBUG command to gain access to the monitor, instead of adding extra instructions to your code.

When the iSDM monitor first receives control, fill the unused portion of the stack with a value that would not normally appear there. For example, use the monitor's S command to fill the remaining stack with a value of 0AAH.

Continue running the program. When the iSDM monitor receives control at the end of the program, examine the stack and see how much of it still contains the value you filled in earlier. That portion was unused throughout the entire execution of the program.

Use this technique to determine stack usage, but do not assume that the value you obtain is an exact number. The value you determine usually won't be exact because a typical run of the program probably will not take the deepest path (using the most stack) through the program. Also, a typical run might not encounter interrupts on the deepest paths through the program.

CHAPTER 5
CONVERTING iRMX® I APPLICATIONS AND IMPROVING PERFORMANCE

5.1 INTRODUCTION

When moving an iRMX I application to the iRMX II Operating System, your first concern is usually making the application work. After you realize that the upward compatibility provided by the iRMX II Operating System makes the change easy, you will want to get optimum performance from the 80286 and 80386 processors. This chapter helps you do that, by providing instructions that will enable you to design applications with optimum performance.

5.2 EFFECTS OF SEGMENTATION

Segmentation is one of the most important performance areas in which the iRMX I and iRMX II Operating Systems differ. When using the 8086 processor, and the 80286/386 processors in real-address mode, changing a segment register is as fast as changing any other register, making it one of the fastest operations to perform. However, with the 80286 and 80386 processors in protected virtual address mode, changing a segment register involves looking up four words in the global or local descriptor tables (GDT or LDT). Therefore, it is a slower operation.

5.2.1 Causes of Program Slowdown

When the processor is in protected virtual address mode (as it is when the operating system runs), it is possible to spend unnecessary time on segment register changes because of the way PL/M-286 works. The default model of segmentation under which PL/M-286 compiles your code is LARGE. Under the LARGE model, the following items apply:

- Every procedure has its own code segment and its own data segment.

- Every reference to a non-local variable causes the processor to set up the ES register to access the variable's segment, an operation requiring 17 clock cycles.
- Every procedure call to an external procedure or to a public internal procedure is a long call, which changes the CS register and costs 26 clock cycles (as compared to 7 cycles for a short call and 13 cycles for a long call in real-address mode). The long call is followed by a MOV to DS to establish a new data segment. This costs an additional 17 clock cycles (as compared to 2 cycles in real-address mode).

The net result of this additional overhead is a significant slowdown when compared to the same instruction running in real-address mode. The overall slowdown of the application depends of course on how often such instructions are used.

5.2.2 Avoiding Slowdown

To help avoid the slowdown caused by changing a segment register, read the "PL/M-286 Extended Segmentation" chapter in the PL/M-286 manual. In addition, use the COMPACT model of PL/M-286 when possible. Use the MEDIUM model when you cannot use COMPACT, and use COMPACT subsystems when you need more speed than MEDIUM can give you. Resulting code will have fewer segment register changes and will run faster. A side benefit of this technique is that it requires fewer objects (the operating system has an upper limit of 8000 objects).

In addition, you can reduce slowdown by using the OPTIMIZE(3) compiler control when compiling code. The *PL/M-286 User's Guide* describes the requirements for including this control. Use the OPTIMIZE(3) control cautiously. If you aren't sure whether your code meets the requirements for OPTIMIZE(3), use OPTIMIZE(2) instead.

5.3 PL/M-286 BASED VARIABLES

Accessing a PL/M-286 variable that is BASED on a POINTER or a SELECTOR requires that the ES register be loaded first. It is therefore a slower operation in protected mode (relative to real mode). To avoid making numerous references to members of a BASED structure or array, you might want to try the following approach:

- Declare a non-BASED copy of the structure or array.
- Use MOVW to copy from the original BASED structure to the non-BASED copy.
- Continue accessing only the copy.

To see if this technique helps, place $CODE and $NOCODE controls around the areas of code in which potential slowdown occurs. This causes the compiler to generate assembly language listings of these areas. With the listings, you can determine how often segment registers are loaded and whether this optimization technique is necessary.

5.4 OPTIMIZING NUCLEUS PERFORMANCE

After you increase your application's speed, try to optimize the use of Nucleus system calls. To do this, you must understand that under the operating system, fast Nucleus system calls get faster and slow system calls get slower, when compared to the iRMX I Nucleus running on a real-mode 80286 processor. Therefore, use the fast system calls (such as sending and receiving messages) as much as possible, and use the slower system calls (creating and deleting) as little as possible.

One way to avoid creating and deleting objects is by reusing segments. Instead of creating a segment each time you need one and deleting it each time you finish, keep the segment for later use. If the segment size is appropriate, you can use it again without incurring the overhead of creating a new segment.

5.5 OPTIMIZING SEQUENTIAL I/O

If your programs use primarily sequential I/O (as opposed to random I/O) you can increase performance when using the Extended I/O System, by assigning larger buffers than you used under the iRMX I Operating System.

5.6 MAILBOX TUNING

When creating a mailbox, you can specify the number of messages that can be waiting in the mailbox's high-performance queue. When this queue overflows, the operating system creates an additional segment to contain 100 more messages. This overflow queue is slower than the high-performance queue only for the initial overflow, because at that time the operating system must create a new segment to serve as the overflow queue (recall that creating a segment is a relatively slow operation). After the overflow queue is created, that queue is just as fast as the high-performance queue. The overflow queue is not deleted until the entire mailbox is empty (both queues).

Because of the mechanism used to create and delete overflow queues, when you create a mailbox give it a queue large enough to handle all the messages you expect to be waiting at the queue. However, if an overflow is possible, you might want to prevent the queue from emptying, so that the operating system won't delete the overflow queue and then be forced to create it again when another overflow occurs.

5.7 RECYCLING BUFFERS

Because creating and deleting segments are slow operations, applications that use many buffers can benefit from reusing buffers from a previously created Buffer Pool instead of creating buffers and deleting them as needed.

You can recycle buffers by creating a Buffer Pool with a sufficient number of buffers during the application's initialization. When you create the Buffer Pool, you have complete control (within limits) over the number and size of the buffers available.

When a buffer is needed a task can invoke the REQUEST$BUFFER system call to receive the token of one of the buffers from the Buffer Pool. This is faster than creating a segment. When the buffer is not needed any more, the task can invoke RELEASE$BUFFER to return the buffer to the Buffer Pool (this is faster than deleting a segment).

5.8 CONSERVING LIMITED RESOURCES

With the iRMX I Operating System, programmers must often pay special attention to the amount of memory in the system. Because the 8086 processor can access only one megabyte of memory, memory is usually the scarcest resource in the system.

Under the iRMX II Operating System, programmers must still pay attention to the amount of memory in the system. But, memory is no longer a scarce resource, because the 80286 and 80386 processors can each access up to their full address range of memory locations. Instead, the most limited resource is the number of entries in the global descriptor table, which serve as tokens for iRMX II objects.

In an iRMX I environment, programs often verify memory problems by examining the condition code returned by each system call and looking for E$MEM conditions. In an iRMX II environment, programs should check for both E$MEM and E$SLOT conditions. The E$SLOT code is returned if there aren't enough GDT slots to complete the requested system call.

Build your iRMX II systems with as many GDT slots as possible. For systems with over 2MB of physical memory, allocating full-size GDT slots (8K slots that consume 128K bytes of memory) seems a reasonable cost for the problems that the extra GDT slots can eliminate.

APPENDIX A
EXAMPLE PROGRAMS

A.1 INTRODUCTION

This appendix contains complete programming examples that use iRMX II system calls. You can compile, bind, and run these programs from the Human Interface yourself, or you can use them purely as examples, to see how to perform certain operations under the operating system

You can also use the EXAMPLE 1 files as a starting point in developing your application code. Using different parts of these files saves you from initially having to create the source module, adding include statements, re-writing code that attaches the console, etc.

A.2 EXAMPLE 1 - SYSTEM PROGRAMMING CONCEPTS

Example 1 demonstrates some iRMX programming concepts by printing prompts to the console screen and accepting input from the user. To accomplish this, the program uses two tasks: the main program code the a second task called TASK2. The main program code is the initial task and creates the second task TASK2.

The function of the main program code is the following:

- set up the programming environment by creating objects, the second task, etc.
- prompt the user for and capture keyboard input
- pass the captured input to TASK2
- exit with an error after receiving three user-supplied keystrokes.

The function of TASK2 is to receive user-supplied keystrokes from the main program code and process them. The processing consists of printing the received keystroke to the screen once every second.

Because the job uses two tasks, each task can perform its function separately from the other task. Communication and data passing between the main program code and TASK2 is handled using some basic iRMX programming techniques.

The following sections show you where to locate the actual code in the iRMX file structure, explain basic iRMX programming concepts used in the example, show how to build the program, and show how to run it.

A.2.1 Program Source Code

The program source code and supporting files can be found in the Intel-supplied iRMX file structure. The complete pathname for the source code files and related files is the following:

```
/rmx286/demo/plm/intro
```

Before attempting to understand this example, you should produce hard copies of the source code files or have easy access to view them from a console screen.

A.2.2 Concepts

This example illustrates nine common iRMX programming concepts. The following list briefly describes each of these concepts:

In-Line Exception Processing	The processing of all errors resulting from iRMX System Calls in your application code rather than using the default exception handler, which deletes tasks that get errors.
Using Literal Files	Using separate files that contain PL/M-286 data structure definitions and literal definitions needed to make system calls. Providing separate literal files, such as these, relieves you from repeating data structure and literal definitions throughout modules.
Getting and Setting Terminal Attributes	Using iRMX System Calls to get the current terminal attributes. After getting and altering the attributes, you can use another iRMX System Call to set them.
Creating Tasks	Using an iRMX System Call to create additional tasks from an existing task.
Cataloging Objects	Describing to the system where key objects the job uses reside. Tasks can easily share cataloged objects.

Using Response Pointers During Inter-Task Communication	Instructing serving tasks where to respond with information that signals the completion of a requesting task. Response Pointers allow serving tasks to keep track of which requesting tasks they are responding to.
Using Buffer Pools	Creating areas of memory for a job that tasks can use as a common memory resource. Once a buffer pool and its buffers have been created, all the system has to do in order to use the memory is to request and release buffers.
Performing Screen Input/Output	Reading and writing data to the physical terminal screen.
Performing Simultaneous Input/Output	Tasks performing I/O operations independent of one another. For example, one task may wait for terminal input while another task processes data and writes it to the terminal.

A.2.2.1 In-Line Exception Processing

In-line exception processing provides a way for your application to handle errors generated from system calls. The flexibility of the iRMX Operating System allows you to determine how to process exceptions: processing them in-line or using the default exception handler. This example demonstrates a method that lets you provide your own exception processing. In order for you to provide your own exception processing, you must do two things: cause the system to pass control to your exception handler routine instead of built-in exception handler routines, and create your own exception handler routine.

To get the operating system to pass control to your routine instead of a built-in routine involves resetting the value of the current task's exception mode and coding your tasks to call your exception handler routine.

This example uses a procedure called SET$EXCEPTION in the file EXCEPT.P28 to reset the exception mode to a value of zero. A value of zero tells the operating system to never pass control to built-in exception handler routines. If you examine the beginning of both the main program code and TASK2, you will see that the very first executable statement is a call to the SET$EXCEPTION procedure as follows:

```
CALL set$exception(NO$EXCEPTIONS);
```

This call passes a zero value parameter (NO$EXCEPTIONS supplied from a literal file) to the procedure. When SET$EXCEPTION executes, it calls GET$EXCEPTION$HANDLER, which returns exception handler information to the data structure addressed by except$info. The procedure then replaces the exception mode with zero using the following statement:

```
except$info.mode = except$mode;
```

The procedure then calls SET$EXCEPTION$HANDLER to reset the exception handler information with the altered data addressed by except$info. By supplying a zero value for mode in the data structure that describes exception information, you tell the system to never pass control to the built-in exception handler routine (refer to the *Extended iRMX II Nucleus Systems Calls Manual* for detailed information on these system calls)

Now that you have got the operating system not to call built-in exception handler routines, you must code your tasks to either check for individual expected errors or to call your own exception handler routine. This example uses a procedure called ERROR$CHECK in the file EXCEPT.P28 as the exception handler routine. Notice that in the source code for the main program code and TASK2, a call to ERROR$CHECK follows every system call. The following code illustrates an example:

```
CALL rq$s$open (co$conn, WRITE$ONLY, 0, @status);
CALL error$check(510,status);
mail$box = rq$lookup$object (CALLER, @(3,'MBX'), INFINITE$WAIT,
                             @status);
CALL error$check(520,status);
pool$tkn = rq$lookup$object (CALLER, @(6,'BUFFER'), INFINITE$WAIT,
                             @status);
CALL error$check(530,status);
```

The above code is from TASK2. Each time a system call is made, a subsequent call is made to ERROR$CHECK passing it a line number and a word containing the status from the previous system call. The routine ERROR$CHECK tests the value of status and returns to the calling task if it is zero (no error occurred). If the value of status is not zero (an error occurred), then ERROR$CHECK builds an error message, prints it to the screen, and exits the job.

NOTE

The line numbers passed as the first parameter in calls to ERROR$CHECK have no implicit meaning. These numbers are simply arbitrary numbers that can be associated with a system call. This technique, allows you to easily find a system call that generates an error.

A.2.2.2 Use of Literal Files

Within the iRMX directory structure, you will find intel-supplied literal files. These files are located in the directory /RMX286/INC and have a file extension of .LIT. Literal files provide many data structure definitions used by iRMX System Calls and some extremely useful literal definitions for PL/M-286 code. Including the appropriate literal files in your program modules saves you the trouble of having to repeatedly enter the data structure definitions manually in your source code and keeps your code independent of changes to these structures between iRMX releases. Because the names used in literal definitions are descriptive, your code becomes easier to understand. It is good programming practice to get in the habit of using Intel-supplied literal files.

This example uses include statements to include various literal files in every source code file. The following PL/M-286 statements are from the main program code in the file DEMO.P28. These statements show how to include six literal files.

```
$include(/rmx286/inc/error.lit)
$include(/rmx286/inc/common.lit)
$include(/rmx286/inc/nstexh.lit)
$include(/rmx286/inc/tscrn.lit)
$include(/rmx286/inc/iaiors.lit)
$include(/rmx286/inc/io.lit)
```

Tables A-1 through A-4 show which Intel-supplied literal files are useful for which types of system calls.

Table A-1 Literal Files Helpful With Nucleus System Calls

Nucleus System Call	Literal File
CREATE$JOB	NSTEXH.LIT
GET$EXCEPTION$HANDLER	NSTEXH.LIT
GET$TASK$TOKENS	NGTTOK.LIT
GET$TYPE	NGTTYP.LIT
SET$EXCEPTION$HANDLER	NSTEXH.LIT

Table A-2 Literal Files Helpful With BIOS System Calls

BIOS System Call	Literal File
AGETCONNECTION$STATUS	IAGTCS.LIT IO.LIT
AGETFILE$STATUS	IAGTFS.LIT IFLTYP.LIT IO.LIT
A$OPEN	IO.LIT
A$PHYSICAL$ATTACH$DEVICE	IO.LIT
A$SEEK	IO.LIT
A$SPECIAL	TSCRN.LIT

Table A-3 Literal Files Helpful With EIOS System Calls

EIOS System Call	Literal File
CREATEIOJOB	NSTEXH.LIT IEXIOJ.LIT
E$CREATE$IO$JOB	NSTEXH.LIT
EXITIOJOB	IEXIOJ.LIT
GET$LOGICAL$DEVICE$STATUS	IO.LIT
LOGICAL$ATTACH$DEVICE	IO.LIT
SGETCONNECTION$STATUS	ISGTCS.LIT IO.LIT
SGETFILE$STATUS	ISGTFS.LIT IFLTYP.LIT IO.LIT
S$OPEN	IO.LIT
S$SEEK	IO.LIT
S$SPECIAL	ISIORS.LIT TSCRN.LIT

Table A-4 Literal Files Helpful With Human Interface System Calls

Human Interface System Call	Literal File
CGETOUTPUT$CONNECTION	HGTOCN.LIT
CGETOUTPUT$PATHNAME	HGTOCN.LIT

Aside from the literal files shown in Tables A-1 through A-4, two other important literal files exist: COMMON.LIT and IAIORS.LIT. COMMON.LIT contains many literal declarations commonly used in PL/M-286 programming. You should include this file in all your PL/M-286 programs. IAIORS.LIT contains the structure for the I/O Result Segment (IORS) returned in most BIOS System Calls. You should include this file in all your PL/M-286 programs that make BIOS System Calls.

A.2.2.3 Getting and Setting Terminal Attributes

Many new users have difficulty understanding how to set the desired attributes for a terminal. Before you set the terminal attributes, you must first get the current attributes by invoking the A$SPECIAL (BIOS) System Call or the S$SPECIAL (EIOS) System Call with the spec$func parameter set to f$get$mode. The main program code illustrates both these calls. The only visible difference between the two calls is the structure of the I/O Return Segment used.

Refer to the main program code in the file DEMO.P28. To set terminal attributes, this code first gets the current terminal attributes by calling A$SPECIAL as follows:

```
CALL rq$a$special (ci$conn, SPECIAL$GET$TERM$DATA, @term$atts,
                   read$mbx, @status);
```

In this call, the literal SPECIALGETTERM$DATA specifies that we are getting current terminal attributes. The pointer @term$atts tells what data structure to place the attribute data into. The token read$mbx indicates the mailbox to which the IORS arrives.

The code then waits indefinitely until the I/O Result Segment arrives. The following statement illustrates how the code waits:

```
iors$tkn = rq$receive$message (read$mbx, INFINITE$WAIT, NIL,
                               @status);
```

The main code then checks to make sure that terminal data did in fact arrive by checking the status of the I/O Result Segment. A zero indicates that the I/O operation was successful.

Because A$SPECIAL and not S$SPECIAL was used to get the terminal information, we must specifically delete the I/O Result Segment. It is okay to delete the I/O Result Segment at this point because the current terminal data now resides in the data structure term$atts. Deleting the I/O Result Segment frees that memory for other uses.

The main program code next modifies two terminal attributes to cause no line editing and no echoing of keystrokes to the screen. The code modifies these attributes using the following code:

```
term$atts.connection$flags = ((term$atts.connection$flags AND
                              (NOT C$MASK$LINE$EDIT)) OR 1) OR
                              C$MASK$ECHO;
```

This long assignment statement effectively uses AND and OR logic to alter the least-significant three bits of the 16-bit connection$flags element of the term$atts data structure. The literals C$MASK$LINE$EDIT and C$MASK$ECHO are equal to 3 and 4, respectively. Refer to the *Extended iRMX II BIOS System Call Manual* for detailed information about A$SPECIAL and the data structure term$atts.

The use of the literals, C$MASK$LINE$EDIT and C$MASKI$ECHO, defined in the literal file TSCRN.LIT, facilitate the setting of the attributes by sparing you from having to type in the correct 16-bit binary sequences. It also makes your program much easier to understand, especially at a future date when you are trying to remember exactly which attributes you were trying to set when you wrote the program.

The main program code uses the S$SPECIAL system call as shown below to write the modified terminal attributes back to the file ci$conn (the physical terminal connection).

```
CALL rq$s$special (ci$conn, SPECIAL$SET$TERM$DATA, @term$atts, NIL,
                   @status);
```

Notice that with the S$SPECIAL call, it is not necessary to specifically delete the I/O Result Segment. The operating system handles the task synchronization (the RQ$RECEIVE$MESSAGE) and the IORS deletion for you.

In summary, to change terminal attributes, you must first get them using A$SPECIAL or S$SPECIAL, alter them, and then set them again using either A$SPECIAL or S$SPECIAL. If you use A$SPECIAL, you must delete the IORS resulting from the I/O operation. If you use S$SPECIAL, the operating system takes care of IORS deletion for you.

NOTE

Although, this example uses both versions of the special call (A$SPECIAL and S$SPECIAL), no reason exists that you could not use one version alone.

A.2.2.4 Creating Tasks

When writing an application, you will normally find that it takes several tasks to accomplish the job. Normally, you code separate tasks in separate files (modules) to ease the maintenance of the tasks. Then, from the main program, you use iRMX System Calls to create task objects for the individual modules the application requires.

In this example, only two tasks exist: the main program code (in the file DEMO.P28) and TASK2 (in the file TASK2.P28). Thus, in the main program code, TASK2 is created and given a priority lower than that of the creating task (the main program code). Regardless of the number of tasks your particular job may have, the principles for task creation remain the same.

The following code shows how the main program code creates and assigns a priority to TASK2:

```
task = rq$create$task ((rq$get$priority (CALLER,@status) - 1),
                       @task2, selector$of(NIL), NIL, 1024,
                       0, @status);
CALL error$check(270,status);
```

Imbedded in the CREATE$TASK System Call is the GET$PRIORITY System Call. The GET$PRIORITY System Call gives the new task a priority one level lower than that of the task creating it (the higher the numeric value the lower the priority). It is also possible to assign an absolute priority if you so desire.

The data segment parameter of the CREATE$TASK call is set to SELECTOR$OF(NIL), which indicates that the task sets up its own data segment.

The stack pointer parameter is set to NIL, which indicates automatic stack allocation. Automatic stack allocation means that the NUCLEUS dynamically creates the stack for the new task. The stack size of the new task has been set to 1024 bytes (the next parameter), which is a fairly arbitrary number. If a task will be making NUCLEUS System Calls, the stack size should be at least 300 bytes. If you assign a stack size which is too small, the operating system informs you with a General Protection fault during the task's execution.

Finally, the task flags parameter has been set to zero, which designates that the new task has no floating-point instructions.

For detailed information on the system calls described in this section, refer to the *Extended iRMX II Nucleus System Calls Reference Manual.*

A.2.2.5 Cataloging Objects

It is possible to catalog the key objects of a job. By cataloging objects, you allow other tasks the ability to look up the objects they need for processing at run time. Also, you give modules a higher degree of independence from other modules. Another useful reason to catalog the key objects is to provide you with a helpful debugging tool. If you are using Soft-Scope 286 or the System Debugger to debug your programs, the advantage of having your objects cataloged is that you have a way to correlate TOKEN values with the names you have assigned in your program. The VIEW DIRECTORY command displays all cataloged objects and their respective character strings.

This example catalogs several objects. The following statements from the main program code in the file DEMO.P28 show how to use iRMX System Calls to catalog the objects described as MBX, SEMAPHORE, BUFFER, and TASK2.

```
CALL rq$catalog$object (CALLER, mail$box, @(3,'MBX'), @status);
CALL error$check(220,status);
semaphore = rq$create$semaphore (0, 3, FIFO$QUEUING, @status);
CALL error$check(230,status);
CALL rq$catalog$object (CALLER, semaphore, @(9,'SEMAPHORE'),
                        @status);
CALL error$check(240,status);
pool$tkn = create$buf$pool(18,18,0,SIZE(buffer),@status);
CALL error$check(250,status);
CALL rq$catalog$object (CALLER, pool$tkn, @(6,'BUFFER'), @status);
CALL error$check(260,status);

/*            By including the line $compact(exports task2)
              in the subsys.inc file we have forced task2 to be
              large model and it therefore creates its own data
              segment. It also prevents us from getting a warning
              when we use @task2 as the pointer to the starting
              address for the new task.
*/
task = rq$create$task ((rq$get$priority (CALLER,@status) - 1),
                        @task2, selector$of(NIL), NIL, 1024, 0,
                        @status);
CALL error$check(270,status);
CALL rq$catalog$object (CALLER, task, @(5,'TASK2'), @status);
CALL error$check(280,status);
```

In each of the above four calls to CATALOG$OBJECT, the first parameter indicates whose job object directory the object will be cataloged under. The token CALLER indicates to use the calling job's (main program code) object directory. The second parameter tells which object is going to be cataloged. The third parameter gives the object a name to be cataloged under. And, the fourth parameter is a pointer that points to the condition code generated by the system call.

For more detailed information about the system call CATALOG$OBJECT, refer to the *Extended iRMX II Nucleus System Calls Reference Manual.*

A.2.2.6 Using a Response Pointer During Inter-Task Communication

When you design an application, tasks need to communicate with one another. For example, a serving task may need to inform a requesting task that a process is done, or that it has received some information. Many times, more than one task needs to communicate with the same task. For example, perhaps a requesting task sends information to several serving tasks, and then needs to be notified when each of the serving tasks are done processing data. iRMX System Calls allow you to SEND and RECEIVE data between tasks and keep track of inter-task communication.

The main program code in the file DEMO.P28 demonstrates how a task sends an object to another task (TASK2). The following code from the main program code shows the iRMX System Call SEND$MESSAGE:

```
DO i = 1 to 3;
    buff$tkn = rq$request$buffer(pool$tkn,size(buffer),@status);
    CALL error$check(290,status);
    buffer = write$read(@message$2, SIZE(message$2), INFINITE$WAIT,
                        @status);
    CALL error$check(300,status);

/*            A semaphore is being passed as
              the exchange to which the response should be sent.
*/

    CALL rq$send$message (mail$box, buff$tkn, semaphore, @status);
    CALL error$check(310,status);
END;

/*   now wait for three responses so that we know the other task
     has processed the data.
*/

units = rq$receive$units (semaphore, 3, INFINITE$WAIT, @status);
CALL error$check(320,status);
bytes$writ = rq$s$write$move (co$conn, @message$3, size(message$3),
                              @status);
CALL error$check(330,status);
```

In the above code, the main program loops three times. Each time through the loop, the main program code sends TASK2 the object buff$tkn. In this example, the object sent is a user-supplied keystroke obtained through the system call WRITE$READ just prior to the SEND$MESSAGE call. After sending the keystroke, the main program code returns to the top of the loop and calls WRITE$READ to wait for another keystroke.

Every time TASK2 receives an object sent from the main program code, it responds by sending a unit to the object SEMAPHORE. TASK2 knows exactly where to send the unit because when the main program code issued the SEND$MESSAGE call, it passed the token for SEMAPHORE to the mailbox. Thus, when TASK2 does the RECEIVE$MESSAGE call, the token for SEMAPHORE can be kept in TASK2's version of the variable SEMAPHORE (SEMAPHORE is not a global variable). This technique makes use of the response and response$ptr parameters for the SEND$MESSAGE and RECEIVE$MESSAGE calls, respectively. By specifying where TASK2 is to respond, the main program code ensures that TASK2 responds to the correct task. In applications where more than two tasks exist, this request/response technique is critical.

TASK2 demonstrates how a task receives objects and can send units to a semaphore, thus ensuring inter-task communication. The following code is from the procedure TASK2 in the file TASK2.P28:

```
buff2$tkn = rq$receive$message (mail$box, INFINITE$WAIT,
                                @semaphore, @status);
    write$mbx = rq$create$mailbox (FIFO$QUEUING, @status);

    DO FOREVER;
        IF status = E$OK THEN
        DO;
            CALL rq$a$write (co$conn, @message, size(message),
                             write$mbx, @status);
            CALL error$check(540,status);
            actual = rq$wait$io (co$conn, write$mbx, INFINITE$WAIT,
                                 @status);
            CALL error$check(550,status);
            IF buff$tkn <> selector$of(NIL) THEN
            DO;
                CALL rq$release$buffer(pool$tkn,buff$tkn,@status);
                CALL error$check(560,status);
            END;

            buff$tkn = buff2$tkn;

            CALL rq$send$units (semaphore, 1, @status);
            CALL error$check(590,status);
        END;
        ELSE IF status <> E$TIME THEN
            CALL error$check(600,status);
```

Using the RECEIVE$MESSAGE call, TASK2 waits at the previously created and cataloged mailbox to receive the object token buff$tkn from the main program code. The second parameter (INFINITE$WAIT) indicates for TASK2 to wait forever until the object arrives. The third parameter (@semaphore) is the response$ptr parameter. This parameter tells the operating system where to put the response token passed in from the SEND$MESSAGE call.

After receiving the object, TASK2 processes it. The processing consists of printing out a message to the console that processing is beginning, performing some housecleaning on the buffer pool (refer to Section A.2.2.7 for information on buffer pools), and updating the unit count for SEMAPHORE.

Updating the unit count for SEMAPHORE is accomplished through the system call SEND$UNITS. In this call, TASK2 sends one unit to the object pointed to by the local variable semaphore. Recall that the local variable semaphore contains the token for the object SEMAPHORE created in the main program code. After updating the unit count, TASK2 begins to print the keystroke repeatedly to the screen, waiting for another object to be received from the main program code.

When the main program code sends the third and final keystroke to TASK2, it examines the number of units in the object SEMAPHORE. The following statement performs this:

```
units = rq$receive$units (semaphore,3,INFINITE$WAIT, @status);
```

The call to RECEIVE$UNITS waits infinitely until three units have arrived at the object SEMAPHORE. When TASK2 has sent over the third unit using SEND$UNITS, the main program code continues processing.

In this example, the processing is immediate and simple. The inter-task communication technique shown here would be very valuable in complex processing such as when one set of data sent to a second task takes longer to process than a second set.

In summary, the system calls SEND$MESSAGE, RECEIVE$MESSAGE, SEND$UNITS, and RECEIVE$UNITS can be used to pass data and provide needed synchronization between tasks. In applications where more than two tasks exist, you can use the response parameter of SEND$MESSAGE and the response$ptr parameter of RECEIVE$MESSAGE to ensure that tasks respond to correct requesting tasks.

A.2.2.7 Using Buffer Pools

Buffer Pools provide your system with an available shared resource of fixed-length segments of memory that can be used, as needed, by any task in your job without having to repeatedly create or delete memory segments. These buffers can be used within a task, passed between tasks, and returned to the Buffer Pool when processing is complete, so as to be available for the next task that needs the memory. Because the allocation of memory is a very slow process relative to other system calls, it is good programming practice to create your Buffer Pools at the beginning of your job. Once the Buffer Pool is created, it is simply a matter of requesting and releasing buffers within the Buffer Pool to make use of the memory.

Both the main program code and TASK2 make use of Buffer Pools. The main program code initially creates the the Buffer Pool and requests buffers. TASK2 releases buffers back to the Buffer Pool. The relationship illustrated between the main program code and TASK2 is one of a supplier task and a consumer task. The main program code is requesting the buffers (supplying) from the Buffer Pool, while TASK2 is releasing (consuming) buffers back to the Buffer Pool.

First, let's look at the code that creates the Buffer Pool. The main program code uses the following code to make a call to the procedure CREATEBUFPOOL:

```
pool$tkn = create$buf$pool(18,18,0,SIZE(buffer),@status);
CALL error$check(250,status);
CALL rq$catalog$object (CALLER, pool$tkn, @(6,'BUFFER'), @status);
CALL error$check(260,status);
```

This call passes parameters that define the buffer pool. The token pool$tkn is used to reference or identify the Buffer Pool. The next two parameters (18 and 18) indicate the maximum number of buffers in the pool and the initial number of buffers. The parameter "SIZE(buffer)" returns the value of one. Thus, the size of each buffer created will by only one byte long. Finally, "@status", is the token that points to the word containing the status of the call.

Notice that after the main program code calls CREATEBUFPOOL, it makes a system call to CATALOG$OBJECT to catalog the newly created Buffer Pool.

The Buffer Pool is actually created in CREATEBUFPOOL in the file CRBPOOL.P28. The following code shows how the Buffer Pool is created using iRMX System Calls:

```
create$buf$pool: PROCEDURE(max_bufs, init_num_bufs, attrs, size,
status_ptr) TOKEN PUBLIC;

DECLARE         /* Parameters */

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

DECLARE         /* Local Parameters */

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

DECLARE         /* Literals */

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

buf_pool = rq$create$buffer$pool(max_bufs, attrs, status_ptr);
CALL error$check(10, status);
DO i = 1 to init_num_bufs;
    buf_tok = rq$create$segment(size, status_ptr);
    CALL error$check(20, status);
    IF status <> E$OK THEN
        RETURN selector$of(NIL);
    CALL rq$release$buffer(buf_pool, buf_tok, BFLAGS, status_ptr);
    CALL error$check(30, status);
    IF status <> E$OK THEN
        RETURN selector$of(NIL);
END;
RETURN buf_pool;

END create$buf$pool;
END crbpool;
```

In order for the procedure CREATEBUFPOOL to create the Buffer Pool, several system calls are used. First, a call to CREATE$BUFFER$POOL is made. This call creates a Buffer Pool in which 18 buffers are allowed and no data chaining between the buffers can occur. Next, the routine loops 18 times (one for each buffer allowed in the Buffer Pool). Each time through the loop, system calls are made to CREATE$SEGMENT and RELEASE$BUFFER. Each time these calls are made, a one-byte memory segment is created and released for use to the Buffer Pool. When the loop finishes, 18 individual one-byte buffers reside in the Buffer Pool ready for use by any task.

In order to use buffers from the Buffer Pool, the main program code and TASK2 must request and release buffers. Recall that when the main program code was involved in its loop to send user-supplied keystrokes to TASK2, that the object being sent was a buffer. Let's look at that code again to see how it requests a buffer from the Buffer Pool and waits for data to arrive in it.

```
DO i = 1 to 3;
    buff$tkn = rq$request$buffer(pool$tkn,size(buffer),@status);
    CALL error$check(290,status);
    buffer = write$read (@message$2, SIZE(message$2),
                         INFINITE$WAIT, @status);
    CALL error$check(300,status);

/*              A semaphore is being passed as
                the exchange to which the response should be sent.
*/

    CALL rq$send$message (mail$box, buff$tkn, semaphore, @status);
    CALL error$check(310,status);
END;
```

The first statement in the loop makes a system call to REQUEST$BUFFER to return token for the buffer. The program then waits indefinitely for the user to enter a keystroke using the WRITE$READ system call. When a key is pressed, the character goes into buffer, which is a variable BASED on buff$tkn. Thus, the character is captured in the first available buffer from the Buffer Pool. The main program code can now proceed to send it to TASK2 through the mailbox MBX.

After TASK2 receives the buffer, it releases the buffer so it is not sitting idle. The code below shows how TASK2 releases buffers from the Buffer Pool:

```
DO;
    CALL rq$a$write (co$conn, @message, size(message), write$mbx,
                     @status);
    CALL error$check(540,status);
    actual = rq$wait$io (co$conn, write$mbx, INFINITE$WAIT,
                         @status);
    CALL error$check(550,status);
    IF buff$tkn <> selector$of(NIL) THEN
    DO;
        CALL rq$release$buffer(pool$tkn,buff$tkn,0,@status);
        CALL error$check(560,status);
    END;

    buff$tkn = buff2$tkn;

    CALL rq$send$units (semaphore, 1, @status);
    CALL error$check(590,status);
END;
```

At this point, TASK2 has already received the buffer at the mailbox. The loop above is performed every time the main program code sends over a keystroke. The first time through the loop, the variable buff$tkn is equal to selector$of(NIL), which is zero. Thus, TASK2 skips around the code that releases the buffer back to the Buffer Pool. The reason this code is skipped is because the character needed is still in the the buffer and TASK2 must get it. The statement "buff$tkn = buff2$tkn" captures the data from the buffer into another area of memory local to this task. TASK2 is now free to later release the buffer since it is really no longer of any use. TASK2 continues processing the keystroke until another keystroke is received.

The second and third times through the loop, TASK2 releases the buffer used during the previous trip through the loop by calling RELEASE$BUFFER. The code releases the buffer before capturing the currently received keystroke. The parameter buff$tkn contains the token that indicates which buffer to release (the same buffer requested by the main program code for the previous loop pass). After releasing the buffer, buff$tkn can then be set equal to buff2$tkn, the token of the buffer containing newly arrived keystroke.

In summary, it is the responsibility of the calling task to create and set up Buffer Pools used between tasks that share data. Once the Buffer Pool has been established, the calling task must request a buffer, get data into it, and pass the buffer to the serving task. After receiving the buffer, the serving task must secure the data and release the buffer back to the Buffer Pool for possible use by other tasks.

For detailed information on the system calls used with Buffer Pools, refer to the *Extended iRMX II Nucleus System Calls Reference Manual.*

A.2.2.8 Methods of Screen Input/Output

During an application that involves keyboards and terminal screens, you will probably find it necessary to be able to read and write information to and from the console screen. This example shows two of three methods that you can use to perform this type of I/O.

A very simple type of I/O is demonstrated through a Human Interface System Call. This call is shown in the file DEMO.P28 in the procedure CLEAR$SCREEN. The following code shows the procedure:

```
clear$screen:   PROCEDURE;

    DECLARE i                   WORD,
            cr$lf$str(*)        BYTE DATA (2,CR,LF);


    DO i = 1 to 25;
        CALL rq$c$send$eo$response (NIL, 0, @cr$lf$str, @status);
    END;

END clear$screen;
```

In the above procedure, the call to C$SEND$EO$RESPONSE is used to clear the screen. This procedure calls C$SEND$EO$RESPONSE 25 times. Each time the call is made, the system sends a carriage return and line feed to the terminal. The parameter NIL indicates that no response is expected back from the user. Thus, the screen is cleared.

This method of I/O is simple and quick. Using this call, you can also cause your program to wait and receive a response from the terminal. To receive a response, provide appropriate values for the first two call parameters.

Refer to the *Extended iRMX Human Interface System Calls Manual* for more information on this system call.

The second and third methods of terminal I/O involve using BIOS System Calls. Both methods also require that you establish connections that can be used to write to and read from the terminal. The following statements from the main program code in the file DEMO.P28 show how to establish these connections:

```
co$conn=rq$s$create$file (@(4,':CO:'), @status);
CALL error$check(100,status);
ci$conn=rq$s$attach$file (@(4,':CI:'), @status);
CALL error$check(110,status);
CALL rq$s$open (co$conn, WRITE$ONLY, 0, @status);
CALL error$check(120,status);
CALL rq$s$open (ci$conn, READ$ONLY, 0, @status);
CALL error$check(130,status);
```

The above code creates a new physical file and its connection for terminal output :CO: and creates a connection to the existing terminal input file :CI:. After creating the connections, the program then opens the :CO: file as WRITE$ONLY and the :CI: file as READ$ONLY. Now, whenever the application needs to perform terminal output, it writes to co$conn. And, whenever the application wants to read from the terminal, it reads from ci$conn.

A second method of I/O that this example shows is demonstrated using the following code from the procedure WRITE$READ in the file DEMO.P28:

```
CALL rq$a$write (co$conn, msg$ptr, msg$size, write$mbx, @status);
CALL error$check(2000,status);
actual=rq$wait$io (co$conn, write$mbx, INFINITE$WAIT, @status);
CALL error$check(2010,status);
```

The above code writes a message to the terminal. The call to A$WRITE sends the message addressed by msg$ptr to the screen. After the I/O is performed, the system returns an I/O Result Segment (IORS) to the mailbox write$mbx. The IORS contains information about the previous I/O call. Next, a call to WAIT$IO is made. This call returns the actual number of bytes written in the previous A$WRITE call. Notice that the time specified to wait for WAIT$IO to return data is indefinite. Thus, when data does arrive, the procedure knows that the I/O operation has completed. The call WAIT$IO also performs one other convenient operation for the programmer -- it recycles the IORS for A$READ, A$WRITE, and A$SEEK calls and deletes the IORS for all other BIOS calls. The user does not have to specifically perform the deletion.

A third method of I/O (not demonstrated in this example) that can be used is very similar to the second example above. This method also involves using read and write system calls to the previously defined files :CO: and :CI:. With this method, however, no subsequent call is made to WAIT$IO. Because no call is made to WAIT$IO, the programmer must wait at the mailbox designated for the IORS, extract information from the mailbox about the I/O operation, and then delete the IORS.

For more information about the calls used to perform terminal I/O, refer to the *Extended iRMX II EIOS System Calls Manual* and the *Extended iRMX II BIOS System Calls Manual.*

A.2.2.9 Simultaneous Input/Output

The final concept that this example demonstrates is simultaneous Input/Output. That is, one task can be performing or waiting for input while another task is processing or performing output.

Once the initial job environment is established, the function of the main program code is to collect three keystrokes from the user. As shown before, the main program code accomplishes this function using the following program loop:

```
DO i = 1 to 3;
    buff$tkn = rq$request$buffer(pool$tkn,size(buffer),@status);
    CALL error$check(290,status);
    buffer = write$read (@message$2, SIZE(message$2),
                         INFINITE$WAIT, @status);
    CALL error$check(300,status);

/*            A semaphore is being passed as
              the exchange to which the response should be sent.
*/

    CALL rq$send$message (mail$box, buff$tkn, semaphore, @status);
    CALL error$check(310,status);
END;
```

As shown above, the main program code waits for the keystroke. If the user does not enter one, the program continues to wait virtually forever. Each time a keystroke is provided, the program "wakes up" and processes it by sending it off to TASK2. After sending the keystroke, the program resumes waiting for another keystroke.

When TASK2 receives the keystroke, it performs some processing such as releasing the buffer the data arrived in and completing its half of some inter-task communication. After this initial processing, TASK2 writes the received keystroke to the terminal at the rate of one character per second. TASK2 continues to write keystrokes until it receives another one. The following code shows how TASK2 indefinitely prints the current keystroke and waits for the next one:

```
CALL rq$a$write (co$conn, build$ptr(buff$tkn,0), 1, write$mbx,
                 @status);
CALL error$check(610,status);
actual = rq$wait$io (co$conn, write$mbx, INFINITE$WAIT, @status);
CALL error$check(620,status);

buff2$tkn = rq$receive$message (mail$box, 100, @semaphore,
                                @status);
```

In the above code, the first four lines accomplish writing the contents of the buffer from the Buffer Pool out to the screen. The last system call to RECEIVE$MESSAGE waits for one second at the mailbox mail$box for the arrival of the next buffer (the next keystroke). If one second elapses, control loops up and eventually returns to the first four statements that again output the keystroke to the screen. If the next buffer arrives at the mailbox before the second elapses, TASK2 begins processing the new keystroke at the top of the infinite loop.

As can be seen from the timing of the I/O from the two tasks, the main program code is clearly waiting for, receiving, and processing keystrokes at the same time that TASK2 is writing the previous keystroke to the terminal and waiting to receive the next one. Thus, input from the terminal and output to the terminal can be occurring in two separate tasks independent of one another.

A.2.3 Compiling and Binding the Code

Along with the source code files are two files you can use to compile and bind the job. The file COMPILE.CSD compiles the source code into object modules. The file DEMO.CSD binds the job into an executable program named EXAMPLE.

To compile the example, enter the Human Interface command SUPER and supply the appropriate password. Running under SUPER gives you all the needed file access rights to run the example. Next, enter the following command from the directory that contains the example's files:

```
-
```

This command executes the submit file COMPILE.CSD. This file initiates the compilation of all the jobs source code files. After compilation, you should have one object file for each source code file in the job.

To bind the example, enter the following command:

```
-
```

This command executes the submit file DEMO.CSD. This file binds the example and places the executable program in the file EXAMPLE in the current default directory.

NOTE

If you wish to generate the example as another user, create new directory, copy the example's files to the new directory, move to that directory, submit COMPILE.CSD, and submit DEMO.CSD. Generating the example from another directory allows you to alter source code, while keeping the original version intact.

A.2.4 Running the Example

You should now have a file called EXAMPLE that you can execute. To run the example, type its name as follows:

After typing the filename, the example prompts you with the following message:

```
iRMX II PL/M Example, V3.0
Copyright 1987/1988 Intel Corporation


Welcome to the PL/M Demo Program!

At the prompt you will be given 60 seconds to hit any key.
If you do not hit a key the demo will continue anyway.
You may hit an "E" if you wish to exit the program.



You now have <xx> seconds left to hit a key.
```

At this point, the example is executing code in the PROMPTANDWAIT procedure in the file DEMO.P28. The example is counting down from 60, waiting for you to press a key to start things going. The string <xx> in the previous screen is the decrementing count. To continue, press a key. After pressing any key, the example clears the screen and prompts you with the following message:

```
Please hit a key which will be forwarded to task2 for processing.
```

Let's assume you enter the letter X for the first "counted" keystroke. The example reads the X from the terminal and passes it on to TASK2. TASK2 "wakes up" and prints out the following message to the screen:

```
TASK2 PROCESSING
Please hit a key which will be forwarded to task2 for processing
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ....
```

The X characters that TASK2 prints to the screen continue to appear at the rate of one per second. The character will repeat indefinitely until you enter another keystroke. Also, notice that the prompt to enter another keystroke is buried in the middle of TASK2's processing message and the string of letters that it displays. A close examination of the main program code and TASK2 show the synchronization used to time the output of these tasks. The tasks use a semaphore to achieve task communication.

Entering the next two keystrokes conclude the example. The following output assumes you enter the characters Y and Z:

```
TASK2 PROCESSING Y
Please hit a key which will be forwarded to task2 for processing
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY ....
TASK2 PROCESSING Z

This concludes the PL/M Demo Program.

This would be a good time to examine the program code to see how
these features work.

We will now exit by generating an error.

INTERNAL ERROR AT # 340 STATUS = 0023: E$SUPPORT
```

After you enter the final keystroke, the main program code recognizes that you have entered three characters. This fact signals the code to end the program. Notice that the main program code ends the program before TASK2 can begin printing the third character to the console screen.

A.3 EXAMPLE 2 - TASK COMMUNICATION

The second example is a simple one that shows one method that two tasks can use to communicate with each other. The example is written in PL/M-286 and can be invoked by the operator from the Human Interface level.

A.3.1 Program Source Code

Figures A-1, A-2, and A-3 list the source code of this example. This example includes three tasks: an initialization task (called INIT) that creates the other two tasks and a mailbox, and two tasks (called ALPHONSE and GASTON) that exchange messages via mailboxes. The next few paragraphs discuss how these tasks operate.

The example runs when invoked at the Human Interface level from the keyboard. The task called INIT runs first, creating a mailbox that it catalogs in the root directory under the name "master." It creates the tasks GASTON and ALPHONSE, and then suspends itself.

When GASTON receives control, it gets the token for the mailbox created by INIT (by looking up the name "master" in the root job's object directory). It then creates a segment (in which it will place a message) and a response mailbox (to which ALPHONSE will send a reply). Next it goes into a loop in which it places a message in the segment (after displaying it on the screen), sends the segment to the master mailbox, and waits at the response mailbox for a reply.

When ALPHONSE receives control, it too gets the token for the mailbox created by INIT (again, by looking up the name in the root job's object directory). It then goes into a loop in which it waits at the mailbox for a message, checks to see if the token it received is a segment, and if it is, places its own message in the segment (after displaying it on the screen), and sends the segment to the response mailbox.

By using the two mailboxes, the tasks ALPHONSE and GASTON are synchronized. GASTON sends a message to the first mailbox and waits at the second one before continuing. ALPHONSE waits at the first mailbox. When it receives a message, it sends a reply to the second mailbox and waits at the first for another message. This cycle continues for 15 messages.

After sending its fifteenth message, GASTON drops out of the loop. Instead of sending a segment to the master mailbox, GASTON displays a final message to the screen and sends the token for a task (the token for the INIT task) to the mailbox. When ALPHONSE receives this token and finds it is not a segment, ALPHONSE drops out of its loop and deletes itself.

To finish the processing, GASTON causes the INIT task to resume processing (remember, the INIT task suspended itself earlier). When INIT takes over, it deletes both offspring tasks and issues an EXITIOJOB system call to return control to the Human Interface level.

In this example, each of the three tasks are contained in separate files. The procedures for compiling and linking assume that source files are called INIT.P28 (shown in Figure A-1), ALPHONSE.P28 (shown in Figure A-2), and GASTON.P28 (shown in Figure A-3).

```
$compact
$debug
init:  DO;

DECLARE token                   LITERALLY           'SELECTOR';
DECLARE fifo                    LITERALLY           '0';
DECLARE E$OK                    LITERALLY           '0';
DECLARE self                    LITERALLY           '0';
DECLARE task$priority           BYTE;
DECLARE calling$task            TOKEN;
DECLARE calling$tasks$job       TOKEN;
DECLARE master$mbox             TOKEN;
DECLARE status                  WORD;
DECLARE init$task$token         TOKEN;
DECLARE gaston$task$token       TOKEN;
DECLARE alphonse$task$token     TOKEN;
DECLARE alphonse$start$add      POINTER;
DECLARE gaston$start$add        POINTER;
DECLARE gaston$ds               WORD EXTERNAL;
DECLARE alphonse$ds             WORD EXTERNAL;
DECLARE stack$pointer           POINTER;
DECLARE stack$size              WORD;
DECLARE task$flags              WORD;

gaston:  PROCEDURE EXTERNAL;
end gaston;

alphonse:  PROCEDURE EXTERNAL;
end alphonse;

$include(/rmx286/inc/nuclus.ext)
$include(/rmx286/inc/eios.ext)

exit:
    PROCEDURE PUBLIC;

    IF status <> E$OK THEN              /* An error has occurred        */
        CAUSE$INTERRUPT(3);

END;                                    /* Exit                         */
```

Figure A-1. Example PL/M-286 Application (INIT)

```
calling$tasks$job = SELECTOR$OF(NIL);    /* Directory in which to     */
                                         /* catalog obj               */
calling$task = SELECTOR$OF(NIL);         /* Task whose priority will  */
                                         /* be gotten                 */
gaston$start$add = @gaston;              /* Set up start addresses    */
                                         /* for tasks                 */
alphonse$start$add = @alphonse;
stack$pointer = NIL;                     /* Values for creating tasks */
stack$size = 1F4H;
task$flags = 0;

init$task$token = RQ$GET$TASK$TOKENS (   /* Get token for init task   */
    self,
    @status);

IF status <> E$OK THEN                   /* An error has occurred     */
    CALL exit;

CALL RQ$CATALOG$OBJECT (                 /* Catalog task token in     */
    calling$tasks$job,                   /* directory of calling      */
    init$task$token,                     /* task's job                */
    @(4,'init'),
    @status);

IF status <> E$OK THEN                   /* An error has occurred     */
    CALL exit;

master$mbox = RQ$CREATE$MAILBOX (        /* Create mailbox tasks use  */
    fifo,                                /* to pass messages.         */
    @status);

IF status <> E$OK THEN                   /* An error has occurred     */
    CALL exit;

CALL RQ$CATALOG$OBJECT (                 /* Catalog mailbox in        */
    calling$tasks$job,                   /* directory of calling      */
    master$mbox,                         /* task's job                */
    @(6,'master'),
    @status);

IF status <> E$OK THEN                   /* An error has occurred     */
    CALL exit;

task$priority = RQ$GET$PRIORITY (        /* Get priority of calling   */
    calling$task,                        /* task                      */
    @status);

IF status <> E$OK THEN                   /* An error has occurred     */
    CALL exit;
```

Figure A-1. Example PL/M-286 Application (INIT) (continued)

```
    task$priority = task$priority + 1;      /* Pick lower priority for   */
                                            /* new tasks                 */
        alphonse$task$token = RQ$CREATE$TASK (
        task$priority,
        alphonse$start$add,
        SELECTOR$OF(@alphonse$ds),
        stack$pointer,
        stack$size,
        task$flags,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    gaston$task$token = RQ$CREATE$TASK(     /* Create tasks              */
        task$priority,
        gaston$start$add,
        SELECTOR$OF(@gaston$ds),
        stack$pointer,
        stack$size,
        task$flags,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    CALL RQ$SUSPEND$TASK(                   /* Suspend self and let      */
        calling$task,                       /* other tasks run           */
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
    CALL exit;

    CALL RQ$EXIT$IO$JOB(
        0,
        NIL,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    LOOP:  GOTO LOOP;                       /* If we got here, we're     */
                                            /* in trouble                */

END;                                        /* Init                      */
```

Figure A-1. Example PL/M-286 Application (INIT) (continued)

```
$compact
$debug
alphonse:  DO;

DECLARE CR                      LITERALLY           '13';
DECLARE LF                      LITERALLY           '10';
DECLARE token                   LITERALLY           'SELECTOR';
DECLARE wait$forever            LITERALLY           '0FFFFH';
DECLARE FOREVER                 LITERALLY           'WHILE 1';
DECLARE self                    LITERALLY           'SELECTOR$OF(NIL)';
DECLARE E$OK                    LITERALLY           '0';
DECLARE calling$tasks$job       TOKEN;
DECLARE master$mbox             TOKEN;
DECLARE response$mbox           TOKEN;
DECLARE status                  WORD;
DECLARE type$code               WORD;
DECLARE time$limit              WORD;
DECLARE count                   WORD;
DECLARE alphonse$ds             WORD PUBLIC;
DECLARE seg$token               TOKEN;
DECLARE seg$size                WORD;
DECLARE display$message(*)      BYTE                DATA (
    CR,LF, 'After you, Gaston', CR, LF);

DECLARE message BASED seg$token STRUCTURE (
                                    count           BYTE,
                                    text(25)        BYTE);
exit:  PROCEDURE EXTERNAL;
end exit;

$include(/rmx286/inc/nuclus.ext)
$include(/rmx286/inc/hi.ext)

alphonse:  PROCEDURE PUBLIC;

    time$limit = 25;                        /* Delay factor for message  */
                                            /* display                   */
    seg$size = 32;                          /* Size of message segment   */
    calling$tasks$job = SELECTOR$OF(NIL);   /* Directory in which to     */
                                            /* look up obj               */

    master$mbox=RQ$LOOKUP$OBJECT (          /* Look up message mailbox   */
        calling$tasks$job,
        @(6,'master'),
        wait$forever,
        @status);
```

Figure A-2. Example PL/M-286 Application (ALPHONSE)

```
IF status <> E$OK THEN                    /* An error has occurred     */
    CALL exit;

DO FOREVER;

    seg$token = RQ$RECEIVE$MESSAGE(       /* Receive Gaston's response */
    master$mbox,
    wait$forever,
    @response$mbox,
    @status);

    IF status <> E$OK THEN                /* An error has occurred     */
    CALL exit;

    type$code = RQ$GET$TYPE(              /* See what kind of object   */
    seg$token,  /* it is */
    @status);

    IF status <> E$OK THEN                /* An error has occurred     */
        CALL exit;

    IF type$code <> 6 THEN                /* If not a segment, stop    */
    DO;
        CALL RQ$SUSPEND$TASK (
        self,
        @status);

        IF status <> E$OK THEN            /* An error has occurred     */
            CALL exit;
    END;

    message.count = size(display$message);
    CALL MOVB(@display$message, @message.text, size(display$message));

    CALL RQ$C$SEND$CO$RESPONSE (          /* Send message to screen    */
        NIL,
        0,
        @message.count,
        @status);

    IF status <> E$OK THEN                /* An error has occurred     */
        CALL exit;

    CALL RQ$SLEEP (                       /* Wait awhile to give user  */
        time$limit,                       /* time to see the message   */
        @status);
```

Figure A-2. Example PL/M-286 Application (ALPHONSE) (continued)

```
        IF status <> E$OK THEN  /* An error has occurred    */
            CALL exit;

        CALL RQ$SEND$MESSAGE (  /* Send message to response */
            response$mbox,  /* mailbox  */
            seg$token,
            SELECTOR$OF(NIL),
            @status);

        IF status <> E$OK THEN  /* An error has occurred    */
            CALL exit;

    END;            /* FOREVER  */
END;                /* Alphonse */
END;
```

Figure A-2. Example PL/M-286 Application (ALPHONSE) (continued)

```
$compact
$debug
gaston:  DO;

DECLARE CR                      LITERALLY           '13';
DECLARE LF                      LITERALLY           '10';
DECLARE token                   LITERALLY           'SELECTOR';
DECLARE fifo                    LITERALLY           '0';
DECLARE self                    LITERALLY           'SELECTOR$OF(NIL)';
DECLARE wait$forever            LITERALLY           '0FFFFH';
DECLARE E$OK                    LITERALLY           '0';
DECLARE parent$task             TOKEN;
DECLARE calling$tasks$job       TOKEN;
DECLARE master$mbox             TOKEN;
DECLARE response$mbox           TOKEN;
DECLARE status                  WORD;
DECLARE time$limit              WORD;
DECLARE count                   WORD;
DECLARE final$count             WORD;
DECLARE gaston$ds               WORD PUBLIC;
DECLARE seg$token               TOKEN;
DECLARE seg$size                WORD;
DECLARE main$message(*)         BYTE                DATA (
    CR,LF, 'After you, Alphonse', CR, LF);
```

Figure A-3. Example PL/M-286 Application (GASTON)

```
DECLARE final$message(*)        BYTE                DATA (
    CR,LF, 'If you insist, Alphonse', CR, LF);

DECLARE message BASED seg$token STRUCTURE(
                                        count       BYTE,
                                        text(27)    BYTE);

exit  PROCEDURE EXTERNAL;
end exit;

$include(/rmx286/inc/nuclus.ext)
$include(/rmx286/inc/hi.ext)

gaston  PROCEDURE PUBLIC;

    count = 0;                              /* Initialize count          */
    final$count = 15;                       /* Set number of loops       */
    time$limit = 25;                        /* Delay factor for display  */
                                            /* to screen                 */
    seg$size = 32;                          /* Size of message segment   */
    calling$tasks$job = SELECTOR$OF(NIL);   /* Directory in which to     */
                                            /* look up obj               */

    master$mbox = RQ$LOOKUP$OBJECT (        /* Look up message mailbox   */
        calling$tasks$job,
        @(6,'master'),
        wait$forever,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    response$mbox = RQ$CREATE$MAILBOX (     /* Create response mailbox   */
        fifo,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    seg$token = RQ$CREATE$SEGMENT(          /* Create message segment    */
        seg$size,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;
```

Figure A-3. Example PL/M-286 Application (GASTON) (continued)

```
DO WHILE count < final$count;
    message.count = size(main$message);
    CALL MOVB(@main$message, @message.text, SIZE(main$message));

    CALL RQ$C$SEND$CO$RESPONSE (          /* Send message to screen   */
        NIL,
        0,
        @message.count,
        @status);

    IF status <> E$OK THEN                /* An error has occurred    */
        CALL exit;

    CALL RQ$SLEEP (                       /* Wait awhile to give user */
        time$limit,                       /* time to see the message  */
        @status);

    IF status <> E$OK THEN                /* An error has occurred    */
        CALL exit;

    CALL RQ$SEND$MESSAGE (                /* Send message to mailbox  */
        master$mbox,
        seg$token,
        response$mbox,
        @status);

    IF status <> E$OK THEN                /* An error has occurred    */
        CALL exit;

    seg$token = RQ$RECEIVE$MESSAGE (      /* Receive response from    */
        response$mbox,                    /* Alphonse                 */
        wait$forever,
        NIL,
        @status);

    IF status <> E$OK THEN                /* An error has occurred    */
        CALL exit;

    count = count+1;
END;                                      /* WHILE                    */

message.count = size(final$message);
CALL MOVB(@final$message,@message.text,SIZE(final$message));

CALL RQ$C$SEND$CO$RESPONSE (              /* Send final message to    */
    NIL,                                  /* screen                   */
    0,
    @message.count,
    @status);
```

Figure A-3. Example PL/M-286 Application (GASTON) (continued)

```
    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    CALL RQ$SEND$MESSAGE (                  /* Send token for mailbox to */
        master$mbox,                        /* mailbox.  This will stop  */
        master$mbox,                        /* other task.               */
        SELECTOR$OF(NIL),
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    parent$task = RQ$LOOKUP$OBJECT (        /* Look up token for calling */
        calling$tasks$job,                  /* task                      */
        @(4,'init'),
        wait$forever,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    CALL RQ$RESUME$TASK(                    /* Resume calling task for   */
        parent$task,                        /* cleanup                   */
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

    CALL RQ$SUSPEND$TASK (                  /* Suspend self just in case */
        self,
        @status);

    IF status <> E$OK THEN                  /* An error has occurred     */
        CALL exit;

END;                                        /* Gaston                    */
END;
```

Figure A-3. Example PL/M-286 Application (GASTON) (continued)

A.3.2 Include Files

As shown in Figures A-1 through A-3, each of the tasks contains $INCLUDE statements to include the external declarations of the iRMX II system calls. INIT.P28 uses both Nucleus and Extended I/O System calls, so it includes the external files for both those layers. ALPHONSE.P28 and GASTON.P28 use Nucleus and Human Interface system calls, so they include the external files for those two layers.

Each task contains its own set of include files because each is a separately compiled module. If the tasks were all contained in the same program module, only one set of $INCLUDE statements would be needed.

A.3.3 Compiling and Binding the Code

The following command is used to compile the three files of PL/M-286 source statements:

The PLM286 commands do not include controls for selecting the model of segmentation (SMALL, COMPACT, MEDIUM, or LARGE) because the $COMPACT control was already included in the source files.

The compiler produces three files of object code. Because the PLM286 command did not specify names for the object code files, the files are given the names INIT.OBJ, ALPHONSE.OBJ, and GASTON.OBJ by default.

After compiling, the object files must be bound together and bound with the iRMX II interface libraries. The BND286 statement used to do this is as follows:

In this BND286 statement, the three object files (INIT.OBJ, GASTON.OBJ, and ALPHONSE.OBJ) are bound together with two libraries: PLM286.LIB and RMXIFC.LIB. PLM286.LIB is the standard PL/M-286 library distributed with the compiler. RMXIFC.LIB is the COMPACT version of the iRMX II interface library.

The OBJECT control specifies the name of the executable file generated by BND286. In this case, the file is called SAMPLE.

The SEGSIZE(STACK(+1500)) control specifies that 1500 bytes of stack should be reserved in addition to the amount required by the program. As listed in Chapter 4, this amount represents the amount required by iRMX II applications that include the Human Interface.

The RCONFIGURE(DYNAMICMEM(5000H)) control directs BND286 to produce an STL (single-task loadable) module and to assign a minimum of 5000H bytes of dynamic memory to the module.

A.3.4 Running the Example

The BND286 command produces an executable file called SAMPLE. To run the example, at the Human Interface prompt type the name of the file.

A.4 EXAMPLE 3 - CONTROL-C HANDLER

This section shows an example of a CONTROL-C handler. It is written in PL/M-286 and uses UDI calls exclusively. It can be invoked from the Human Interface level.

A.4.1 Source Code

Figure A-4 contains the source code for the CONTROL-C example (a file called ECHO.P28). The main program echoes to the screen any line entered at the terminal. However, before starting this infinite loop, the program installs a CONTROL-C handler and opens a connection to the terminal. The CONTROL-C handler (a procedure called CC$TRAP) is invoked when the operator enters a CONTROL-C at the keyboard. It resets the CONTROL-C handler, issues a message to the terminal, and exits to the Human Interface.

```
$large DEBUG PW(79) ROM
ECHO DO;
$INCLUDE(/RMX286/INC/RMXPLM.EXT)

DECLARE
    NUL                 LITERALLY       'SELECTOR$OF(NIL)',
    JOB$ABORTED         LITERALLY       '4',
    CR                  LITERALLY       '0DH',
    LF                  LITERALLY       '0AH',
    READ$WRITE          LITERALLY       '3',
    TRUE                LITERALLY       '0FFH',
    FALSE               LITERALLY       '0';

DECLARE
    CONSOLE             TOKEN,
    ACTUAL              WORD,
    STATUS              WORD,
    IO$BUFFER( 256 )    BYTE;
DECLARE
    INIT$MSG( * )  BYTE  DATA(CR,LF,'ENTER DATA FROM KEYBOARD',CR,LF),
    CONTROL$C$MSG( * )  BYTE  DATA(CR,LF,'GONNA GO BYE BYE NOW',CR,LF);

CC$TRAP:  PROCEDURE     PUBLIC;
DECLARE
    STATUS      WORD;
/*
    Let the Operator know that we are finished.
*/
CALL DQ$WRITE(
    CONSOLE,
    @CONTROL$C$MSG,
    SIZE(CONTROL$C$MSG),
    @STATUS);
/*
    Return control back to the CLI
*/
CALL DQ$EXIT( JOBABORTED );

END CC$TRAP;
```

Figure A-4. CONTROL-C Handler Example

```
/*
    Main line code is required so the Application Loader gets an
    initialization record.  We will open a connection to the terminal
    and set a CONTROL-C handler in place before installing the interrupt
    handler.  If a CONTROL-C is entered from the keyboard the CONTROL-C
    handler, CC$TRAP, will reset the interrupt and then call DQ$EXIT
*/

/*
    Install the CONTROL-C handler
*/
CALL DQ$TRAP$CC(
    @CC$TRAP,
    @STATUS);
/*
    Create an open connection to the console. :CI: and :CO: will
    automatically be cataloged in our job's object directory as part
    of the loading process.
*/
CONSOLE = DQ$ATTACH(
    @(4,':CO:'),
    @STATUS);
CALL DQ$OPEN(
    CONSOLE,
    READ$WRITE,
    0,
    @STATUS);
/*
    Inform the Operator.
*/
CALL DQ$WRITE(
    CONSOLE,
    @INITMSG,
    SIZE(INIT$MSG),
    @STATUS);

DO WHILE TRUE;
/*
    Enter forever loop echo lines entered at the console.
*/
    /*
        Read a line from the keyboard.
    */
    ACTUAL = DQ$READ(
        CONSOLE,
        @IO$BUFFER,
        SIZE(IO$BUFFER),
        @STATUS);
```

Figure A-4. CONTROL-C Handler Example (continued)

```
    /*
        Echo it back to the crt.
    */
    CALL DQWRITE(
        CONSOLE,
        @IO$BUFFER,
        ACTUAL,
        @STATUS);
    END; /* WHILE 1 */

END ECHO;
```

Figure A-4. CONTROL-C Handler Example (continued)

A.4.2 Compiling and Binding the Code

The following command is used to compile the code for this example:

The PLM286 command doesn't include controls for selecting the model of segmentation (SMALL, COMPACT, MEDIUM, or LARGE) because the $LARGE control was already included in the source files. The ROM control is also included, to place the constants in the data segment.

The compiler places the generated object code into the file ECHO.OBJ.

After compiling, the object file must be bound with the appropriate iRMX II interface libraries. The BND286 statement used to do this is as follows:

A.4.3 Running the Example

The BND286 command produces an executable file called ECHO. To run the application, simply type the name of the file at the Human Interface prompt. To invoke the CONTROL-C handler and stop the program, hold down the CONTROL key and type C.

A.5 HARDWARE REQUIREMENTS FOR REMAINING EXAMPLES

Sections A.6 and A.7 show examples of interrupt handlers. Both of these examples require hardware that might not be available on all iRMX II systems. This hardware includes an iSBX 350 parallel MULTIMODULE board and an iCS 920 digital signal termination panel. The examples were developed using the iSBX 350 module mounted on an iSBC 286/10A processor board.

Before running the examples in sections A.6 and A.7, you should configure the hardware as follows:

iSBX 350 configuration

- Add jumper E5-E6, placing +5V on J1 pin 50
- Remove jumper E1-E2 and add jumper E2-E3. This configures Port A for input.
- Remove jumper E13-E14 and add jumper E13-E18. This configures the Port A interrupt to MINTR0.
- Install the iSBC 901 terminator packs in sockets XU3, XU5, and XU6.
- Install the 7438 buffer in socket XU4.

iCS 920 configuration

- Connect jumper E2-E3, enabling the +5V from the iSBX 350 to power the iCS 90 board.
- For wirewrap matrices J1 through J24, install jumper 1-2, enabling the LED indicators. Also install jumper 3-4 to wire the iSBX signals to the plus connectors on the terminal strip. Finally, install jumper 5-6 to place the ground to the connectors on the terminal strip.

Cable configuration

- Connect the iSBX 350 board to the iCS 920 board with a 50-pin ribbon cable (3M 3365-50 or equivalent). When fabricating the ribbon, note that the iCS 920 connector J25 is wired the opposite of the iSBX 350 pins.
- Connect the cable to the iSBX 350 board using a connector equivalent to one of the following:

  3M 3415-0000
  AMP 88083-1
  ANSLEY 609-5015
  SAE SD6750 Series

- Connect the cable to the iCS 920 board using a connector equivalent to one of the following:

  3M 3425-7050
  ANSLEY 609-5001M

Switches

- Pull-up resistors are provided to +5V on both the iCS 920 and iSBX 350 boards. Single-pole, single-throw switches will be adequate for data inputs 0 through 7. The interrupt switch should be a momentary-contact switch.
- Switches 0 through 7 should be wired to terminals A0 through A7 respectively. The momentary-contact switch should be wired to terminal C4.

NOTE

Because timed interrupt waits are available with the Operating System, testing can proceed without the interrupt circuit in place.

iSBC 286/10A configuration

- Install the iSBX 350 module in the J5 connector of the iSBC 286/10A board.

A.6 EXAMPLE 4--INTERRUPT TASK

This example shows an interrupt task that works with the hardware described in the previous section. In this example, the interrupt task echoes the Port A switches to the Port B lights whenever the interrupt switch is toggled.

This example is invoked from the Human Interface level. For the example to work, the user must have a maximum task priority of 126 or higher (numerically lower). The example also includes a CONTROL-C handler, enabling you to cancel the program and return to the Human Interface level by entering a CONTROL-C.

A.6.1 Source Code

Figure A-5 contains the source code for the interrupt task example (a file called INTRTSK.P28). The program consists of an interrupt handler, a CONTROL-C handler, and a main program.

The interrupt handler (SBX350INTHNDLR) receives control whenever the interrupt switch is toggled. It informs the interrupt task (the main program) of the interrupt by issuing the SIGNAL$INTERRUPT system call. It performs no other operations.

The CONTROL-C handler (CC$TRAP) is invoked whenever the operator enters a CONTROL-C at the keyboard. When this happens, the routine invokes the RESET$INTERRUPT system call to reset the CONTROL-C handler and then exits to the Human Interface level by invoking the DQ$EXIT system call.

The main program issues a DQ$TRAP$CC system call to set up the CONTROL-C handler. Then it creates a connection to the terminal and initializes the iSBX 350 interface. When this is complete, it issues the SET$INTERRUPT system call to install the interrupt handler. Finally, it enters a loop where it performs a WAIT$INTERRUPT to wait for the interrupt signal from the interrupt handler. When it receives notification of the interrupt, it reads the status of the switches and sets the lights accordingly. Then it waits for the next interrupt.

```
$large DEBUG PW(79) ROM
SBX350_MOD:  DO;
/*
WARNING:
    In order to run this lab from the Human Interface the user must
    have the ability to create tasks with a priority of 126.  If you
    are using dynamic logon terminals the process involves creating
    a user with a max priority of 126 or less.  If you are using a
    static logon terminal, you may have to reconfigure your system.
*/
$INCLUDE(/RMX286/INC/RMXPLM.EXT)

DECLARE
    NUL                     LITERALLY       'SELECTOR$OF(NIL)',
    /*
        Use MINTR0 from an iSBX 350 installed on J5
    */
    SBX350$INTR$LEVEL       LITERALLY       '075H',
    JOB$ABORTED             LITERALLY       '4',
    CR                      LITERALLY       '0DH',
    LF                      LITERALLY       '0AH',
    READ$WRITE              LITERALLY       '3',
```

Figure A-5. Interrupt Task Example

```
    SBX350$PORT$A           LITERALLY       '0A0H',
    SBX350$PORT$B           LITERALLY       '0A2H',
    SBX350$PORT$C           LITERALLY       '0A4H',
    SBX350$CONTROL$PORT     LITERALLY       '0A6H',
    /*
        Set Port A for strobed input and Port B for Mode 0 output
    */
    SBX350$INIT$MODE        LITERALLY       '10111000B',
    ENABLE$PORT$A$INTERRUPT LITERALLY       '00001001B';

DECLARE
    (
    CONSOLE     )       TOKEN,
    (
    STATUS  )           WORD;
DECLARE
    SBXINITMSG  (*) BYTE    DATA    (
    CR,LF,'THE INTERRUPT TASK IS NOW INITIALIZED',CR,LF),
    CONTROL$C$MSG   (*) BYTE    DATA    (
    CR,LF,'GONNA GO BYE BYE NOW',CR,LF);

CC$TRAP:  PROCEDURE PUBLIC;
DECLARE
    STATUS  WORD;
/*
    Clear the interrupt.
*/
CALL RQ$RESET$INTERRUPT(
    SBX350$INTR$LEVEL,
    @STATUS );
/*
    Let the Operator know that we are finished.
*/
CALL DQ$WRITE(
    CONSOLE,
    @CONTROL$C$MSG,
    SIZE(CONTROL$C$MSG),
    @STATUS);
/*
    Return control back to the CLI
*/
CALL DQ$EXIT(
    JOBABORTED );

END CC$TRAP;
```

Figure A-5. Interrupt Task Example (continued)

```
SBX350$INT$HNDLR:  PROCEDURE    INTERRUPT   PUBLIC;

DECLARE STATUS WORD;

/*
    Let the interrupt task do all of the work.  The time for a
    context switch should help in the debounce of the interrupt
    switch.
*/
CALL RQ$SIGNAL$INTERRUPT( SBX350$INTR$LEVEL, @STATUS );

END     SBX350$INT$HNDLR;

/*
    Main line code is require so the application loader gets an
initialization record.

    We will open a connection to the terminal and set a CONTROL-C
    handler in place before installing the interrupt handler.

    If a CONTROL-C is entered from the keyboard, the CONTROL-C
    handler, CC$TRAP, will reset the interrupt and then call DQ$EXIT
*/

/*
    Install the CONTROL-C handler
*/
CALL DQ$TRAP$CC(
    @CC$TRAP,
    @STATUS);
/*
    Create an open connection to the console.
*/
CONSOLE = DQ$ATTACH(
    @(4,':CO:'),
    @STATUS);
CALL DQ$OPEN(
    CONSOLE,
    READ$WRITE,
    0,
    @STATUS);
```

Figure A-5. Interrupt Task Example (continued)

```
/*
    Initialize the SBX350 interface.
    Port A = Input
    Port B = Output
    Port C = Output
    Port A interrupt enabled
*/
OUTPUT(SBX350$CONTROL$PORT) = SBX350$INIT$MODE;
OUTPUT(SBX350$CONTROL$PORT) = ENABLE$PORT$A$INTERRUPT;
/*
    Install the Interrupt Handler.
*/
CALL RQ$SET$INTERRUPT(
    SBX350$INTR$LEVEL,
    1,
    @SBX350$INT$HNDLR,
    NUL,
    @STATUS);
/*
    Inform the Operator.
*/
CALL DQ$WRITE(
    CONSOLE,
    @SBXINITMSG,
    SIZE(SBX$INIT$MSG),
    @STATUS);

/*
    Enter forever loop waiting for interrupts.
*/
DO WHILE 1;
        CALL RQ$WAIT$INTERRUPT(
            SBX350$INTR$LEVEL,
            @STATUS );

    /*
        Wait to debounce the interrupt switch.
    */
    CALL RQ$SLEEP(
        1,
        @STATUS);
    /*
        Get the input data.
    */
    OUTPUT( SBX350$PORT$B ) = NOT (INPUT( SBX350$PORT$A ));
    END; /* WHILE 1 */

END SBX350_MOD;
```

Figure A-5. Interrupt Task Example (continued)

A.6.2 Compiling and Binding the Code

The following command is used to compile the code for this example:

The PLM286 command doesn't include controls for selecting the model of segmentation (SMALL, COMPACT, MEDIUM, or LARGE) because the $LARGE control was already included in the source files. The ROM control is also included, to place the constants in the data segment.

The compiler places the generated object code into the file INTRTSK.OBJ.

After compiling, the object file must be bound with the appropriate iRMX II interface libraries. The BND286 statement used to do this is as follows:

A.6.3 Running the Example

The BND286 command produces an executable file called INTRTSK. To run the applications, type the name of the file at the Human Interface prompt. To invoke the CONTROL-C handler and stop the program, hold down the CONTROL key and type C.

A.7 EXAMPLE 5--FIRST-LEVEL JOB

This section contains a larger, more complicated example that is typical of a real-world application. It is a multitasking, data acquisition application configured as a first-level job. Each time an interrupt occurs from the iCS 920 board, the application reads the iCS 920 switches via the iSBX 350 board installed on the iSBC 286/10A board. After a number of interrupts, the application calculates the mean, median, and mode of the inputs and and displays this information at the terminal. The application also accepts input from the terminal to indicate how many samples to take before calculating the results.

The application is divided into two jobs: a first-level job that handles all the data acquisition and formats messages for the terminal, and a terminal job that handles all I/O with the terminal. The terminal job is an offspring of the first-level job.

Within the first-level job are three tasks: an initialization task (INITTSK), an interrupt task (SBX350TSK), and a process task (PROCESSTSK). The initialization task creates the other tasks in the job and also creates the offspring job. The interrupt task collects the data from the iSBX350 board and passes it on to the process task. The process task receives instructions from the I/O job and sends formatted information to the I/O job.

The terminal job also contains three tasks: an initialization task (TERMINITTSK), a terminal input task (TERMINTSK), and a terminal output task (TERMOUTTSK). The initialization task creates the other tasks in the job and establishes a connection to the terminal. The terminal input task reads input from the terminal. The terminal output task sends output to the terminal.

The tasks in this application communicate via mailboxes. The interrupt task creates and catalogs a mailbox called SBX350MBX where it will receive data from the process task. It also looks up the name of the process task's mailbox, which it uses to send information to the process task. Likewise, the process task creates a mailbox called PROCIOMBX and looks up the name of the interrupt task's mailbox. In addition, the process task looks up the names of the mailboxes created by the terminal input and terminal output tasks so that it can send data to and receive data from the terminal. These mailboxes, called TERMINMBX and TERMOUTMBX are created by the terminal input and terminal output tasks when they start running.

A.7.1 Source Code

The source code for this example is divided into five separate files. INITTSK.P28 contains the initialization task for the first-level job (INITTSK). It is listed in Figure A-6.

INITTSK performs the following operations:

- Creates the interrupt task SBX350TSK
- Creates the process task PROCESSTSK
- Creates the terminal job

TERM.P28 contains all the tasks in the terminal job (TERMINITTSK, TERMINTSK, and TERMOUTTSK). It is listed in Figure A-7.

TERMINITTSK performs the following operations:

- Creates a connection to the terminal and catalogs it as CONSOLE
- Creates the terminal input and output tasks

TERMINTSK performs the following operations:

- Looks up the connection for CONSOLE
- Creates the mailbox for terminal input and catalogs it as TERMINMBX

- Starts an infinite loop in which it waits at the TERMINMBX mailbox for an input request, reads a line of input, and sends the input data to the response mailbox specified

TERMOUTTSK performs the following operations:

- Looks up the connection for CONSOLE
- Creates the mailbox for terminal output and catalogs it as TERMOUTMBX
- Starts an infinite loop in which it waits at the TERMOUTMBX mailbox for an output message, formats the message and sends it to the console, and deletes the message

SBX350.P28 contains the code for the interrupt task (SBX350TSK) and the interrupt handler (SBX350INTHNDLR). This file is listed in Figure A-8.

SBX350TSK performs the following operations:

- Creates and catalogs the mailbox SBX350MBX.
- Initializes the iSBX 350 board.
- Sets the SBX350INTHNDLR procedure as the interrupt handler associated with this task.
- Looks up the TERMOUTMBX mailbox and sends an initialization message there.
- Looks up the PROCIOMBX mailbox and sends initialization data there.
- Begins an infinite loop in which it waits at the SBX350MBX mailbox for a message indicating the number of samples to take. For each sample, it waits for an interrupt, obtains the sample from the iSBX 350 board, stores the data in an array, and clears the interrupt. When it has obtained all the samples, it sends the data to the PROCIOMBX.

SBX350INTHNDLR simply invokes the SIGNAL$INTERRUPT system call. This passes control to the SBX350TSK interrupt task, which actually handles the interrupt.

PROCIO.P28 contains the code for the process task (PROCESSTSK). This file is listed in Figure A-9.

PROCESSTSK performs the following operations:

- Looks up the TERMINMBX, TERMOUTMBX, and SBX350MBX mailboxes.
- Creates a mailbox for communication with SBX350TSK and catalogs it as PROCIOMBX. It waits at the mailbox to synchronize with SBX350TSK.
- Sends a parameter request message to TERMOUTMBX, sends an input request message to TERMINMBX, waits at the response mailbox for the terminal input, and verifies the input as valid. PROCESSTSK continues to perform this step until it receives valid input. Then it sends the valid message to the SBX350MBX mailbox.

- Begins an infinite loop in which the following operations are performed:
  - Waits at the PROCIOMBX for data from the interrupt task.
  - Formats the data and sends it to the TERMOUTMBX mailbox.
  - If no input request is pending, sends a message to the TERMINMBX mailbox requesting input.
  - Checks the response mailbox without waiting to see if the operator has responded with an input request.
  - If there is an input request, validates it and passes it to the interrupt task via the SBX350MBX mailbox. Then deletes the segment containing the data.

UTILS.P28 contains utility procedures used by the various tasks. The procedures include STRLEN to return the length of a null-terminated string, BTOA to take a byte value and convert it to ASCII, ATOB to convert an ASCII string to a byte value, CALC$MEAN to calculate the mean, CALC$MEDIAN to calculate the median, and CALC$MODE to calculate the mode. This file is listed in Figure A-10.

```
$COMPACT DEBUG PW(79) ROM
STARTUP:  DO;

$INCLUDE(/RMX286/INC/RMXPLM.EXT)

DECLARE
    NUL                     LITERALLY       'SELECTOR$OF(NIL)',
    NO$NPX                  LITERALLY       '0',
    ALL$ERRORS              LITERALLY       '3';

SBX350_TSK:  PROCEDURE  EXTERNAL;
END SBX350_TSK;

TERM_INIT_TSK:  PROCEDURE  EXTERNAL;
END TERM_INIT_TSK;

PROCIO_TSK:  PROCEDURE  EXTERNAL;
END PROCIO_TSK;

/*
    Setup public data variable so this can be configured in as
    a first level job.
*/
DECLARE     DUMMY$DATA$VARIBLE  BYTE    PUBLIC;

INIT_TSK:  PROCEDURE    PUBLIC;
/*
    This task is the initial task of our first level job.  It
    creates two other tasks:  BX350TSK, which will become the
    interrupt task handling digital input, and PROCIOTSK, which
    is responsible for processing the the input data and
    providing formatted I/O for the operator.  This task also
    create a separate job for terminal I/O.
*/
DECLARE

    TERM$JOB$TOKEN          TOKEN,
    INIT$JOB$TOKEN          TOKEN,
    SBX350$TASK$TOKEN       TOKEN,
    PROCIO$TASK$TOKEN       TOKEN,
    STATUS                  WORD,
    DUMMY$EH$STRUC          STRUCTURE   (
                                EH$PTR          POINTER,
                                EH$MODE         BYTE);

CAUSEINTERRUPT(3);
```

Figure A-6. First-Level Job Initialization Task (INITTSK.P28)

```
/*
    Set up the exception handler structure.  PLM 286 does not
    allow NIL to be used within a DATA declaration.  Setup of
    DUMMY$EH$STRUC must be done dynamically.  Note:  We could
    have lied to PLM by declaring EH$PTR as a DWORD.
*/
DUMMY$EH$STRUC.EH$PTR = NIL;
DUMMY$EH$STRUC.EH$MODE = ALL$ERRORS;

/*
    Create the task which handles digital input from the SBX350.
    Note:  This will become an interrupt task.
*/
SBX350$TASK$TOKEN = RQ$CREATE$TASK(
    139,
    @SBX350_TSK,
    SELECTOR$OF(@DUMMY$DATA$VARIBLE),   /* Initialize the DS, as we are  */
                                        /* in COMPACT                    */
    NIL,                                /* Let rmx assign the stack      */
    400H,                               /* Stack Size                    */
    NO$NPX,                             /* This task doesn't use the NPX */
    @status );

/*
    Create the process control task that manipulates the digital data.
*/
PROCIO$TASK$TOKEN = RQ$CREATE$TASK(
    139,
    @PROCIO_TSK,
    SELECTOR$OF(@DUMMY$DATA$VARIBLE),   /* Initialize the DS, as we are  */
                                        /* in COMPACT                    */
    NIL,                                /* Let rmx assign the stack      */
    400H,                               /* Stack Size                    */
    NO$NPX,                             /* This task doesn't use the NPX */
    @status );

/*
    Create the terminal job to process console input.  The
    terminal init task will create an input task and an output
    task.  A job is not actually required for this application
    but we are doing it to show RQE$CREATE$JOB.
*/
INIT$JOB$TOKEN = RQ$GET$TASK$TOKENS(
    1,                                  /* Get this job's token.         */
    @STATUS );
```

Figure A-6. First-Level Job Initialization Task (INITTSK.P28) (continued)

```
TERM$JOB$TOKEN = RQ$E$CREATE$JOB(
    40H,                /* We will be cataloging objects in this        */
                        /* job's directory                              */
    INIT$JOB$TOKEN,     /* Pass this job's token to the terminal job    */
    400H,               /* Minimum memory pool                           */
    0FFFFFH,            /* Memory pools can contain up to 16 Megabytes. */
    0100H,              /* Maximum objects for this job                 */
    3,                  /* TERMINITTSK plus two others                  */
    0,                  /* No limit on priority of tasks in this job    */
    @DUMMY$EH$STRUC,    /* Use SDB as error handler                     */
    0,                  /* Require  parameter checking as this job      */
                        /* makes BIOS calls.                            */
    139,                /* Initial task priority is 139                 */
    @TERM_INIT_TSK,
    SELECTOR$OF(@DUMMY$DATA$VARIBLE),   /* Initialize the DS, as we are */
                                        /* in COMPACT                   */
    NIL,                /* Let rmx assign the stack                     */
    400H,               /* Stack Size                                   */
    NO$NPX,             /* This task does not use the NPX               */
    @status );
/*
    Initialization is complete, so delete this task
*/
CALL RQ$DELETE$TASK(
    NUL,
    @STATUS );

END INIT_TSK;

END     STARTUP;
```

Figure A-6. First-Level Job Initialization Task (INITTSK.P28) (continued)

```
$compact DEBUG pw(79) ROM
TERM_MOD:  DO;

$INCLUDE(/RMX286/INC/RMXPLM.EXT)

DECLARE
    NUL                          LITERALLY       'SELECTOR$OF(NIL)',
    CONNECTION$OBJECT$TYPE       LITERALLY       '0101H',
    PARAMETER$OBJECT$TYPE        LITERALLY       '02',
    WAIT$FOREVER                 LITERALLY       '0FFFFH',
    PHYSICAL$FILE$DRIVER         LITERALLY       '01',
    READ$WRITE                   LITERALLY       '3',
    SHARE$ALL                    LITERALLY       '3',
    DRAU                         LITERALLY       '00001111B',
    SUPER                        LITERALLY       '0',
    WORLD                        LITERALLY       '0FFFFH',
    NO$NPX                       LITERALLY       '0',
    TRUE                         LITERALLY       '0FFH',
    FALSE                        LITERALLY       '0',
    CR                           LITERALLY       '0AH',
    LF                           LITERALLY       '0DH';

STRLEN:  PROCEDURE( STRPTR )     BYTE            EXTERNAL;
    DECLARE STRPTR  POINTER;
END STRLEN;

TERMIN_TSK:  PROCEDURE  PUBLIC;
/*
    TERMINTSK receives messages at TERMINMBX.  It performs a
    read operation and returns a zero terminated string to the
    response mailbox specified during the receive message.
*/
DECLARE
    PARAMETER$OBJECT$TOKEN       TOKEN,
    TERM$IN$MBX                  TOKEN,
    TERM$OUT$MBX                 TOKEN,
    CONSOLE$TOKEN                TOKEN,
    BUFFERT                      TOKEN,
    USER$RSP$MBX                 TOKEN,
    BIOS$RSP$MBX                 TOKEN,
    BUFFER  BASED                BUFFERT (128)   BYTE,
    ACTUAL                       WORD,
    STATUS                       WORD;
```

Figure A-7. Terminal Job (TERM.P28)

```
/*
    Create the mailbox at which keyboard input requests will be received.
*/
TERMINMBX = RQ$CREATE$MAILBOX(
        0,
        @STATUS );
/*
    Catalog the mailbox as "TERMIN" in the parent job.
*/
PARAMETER$OBJECT$TOKEN = RQ$GET$TASK$TOKENS(
    PARAMETER$OBJECT$TYPE,
    @STATUS );
CALL RQ$CATALOG$OBJECT(
    PARAMETER$OBJECT$TOKEN,
    TERM$IN$MBX,
    @(6,'TERMIN'),
    @STATUS );
/*
    Lookup the open connection for the keyboard
*/
CONSOLE$TOKEN = RQ$LOOKUP$OBJECT(
    NUL,
    @(7,'CONSOLE'),
    WAIT$FOREVER,
    @STATUS );
/*
    Create the response mailbox to be used with BIOS system calls.
*/
BIOS$RSP$MBX = RQ$CREATE$MAILBOX(
    0,
    @STATUS );
DO WHILE TRUE;
/*
    Enter forever loop waiting for input requests.
*/
    /*
        Wait for an input request.
    */
    BUFFERT = RQ$RECEIVE$MESSAGE(
        TERM$IN$MBX,
        WAIT$FOREVER,
        @USER$RSPMBX,
        @STATUS );
```

Figure A-7. Terminal Job (TERM.P28) (continued)

```
    /*
        Send a read request to the keyboard.
    */
    CALL RQ$A$READ(
        CONSOLE$TOKEN,
        @BUFFER,
        SIZE(BUFFER),
        BIOS$RSP$MBX,
        @STATUS );
    /*
        Wait for a <CR> input from the keyboard.
    */
    ACTUAL = RQ$WAIT$IO(
        CONSOLE$TOKEN,
        BIOS$RSP$MBX,
        WAIT$FOREVER,
        @STATUS );
    /*
        The terminal driver will append a <LF> after the <CR>.
        We don't want either of them so zero out the next to last
        character.
    */
    IF ACTUAL >= 2 THEN
        BUFFER( ACTUAL - 2 ) = 0;
    ELSE
        BUFFER( 0 ) = 0;
    /*
        Send the zero terminated string back to whomever requested it.
    */
    CALL RQ$SEND$MESSAGE(
        USER$RSPMBX,
        BUFFERT,
        NUL,
        @STATUS );
    END; /* DO WHILE 1 */

END TERMIN_TSK;
```

Figure A-7. Terminal Job (TERM.P28)
(continued)

```
TERMOUT_TSK:  PROCEDURE PUBLIC;
/*
    TERMOUTTSK expects to receive zero terminated strings at TERMOUTMBX.
    It will then write these string out to the CONSOLE.
*/
DECLARE
    PARAMETER$OBJECT$TOKEN      TOKEN,
    TERM$OUT$MBX                TOKEN,
    CONSOLE$TOKEN               TOKEN,
    BUFFERT                     TOKEN,
    USER$RSP$MBX                TOKEN,
    BIOS$RSP$MBX                TOKEN,
    BUFFER  BASED   BUFFERT (1) BYTE,
    ACTUAL                      WORD,
    STATUS)                     WORD;

/*
    Create the mailbox at which console output messages will be received.
*/
TERMOUTMBX = RQ$CREATE$MAILBOX(
        0,
        @STATUS );
/*
    Catalog the mailbox as "TERMOUT" in the parent job.
*/
PARAMETER$OBJECT$TOKEN = RQ$GET$TASK$TOKENS(
    PARAMETER$OBJECT$TYPE,
    @STATUS );
CALL RQ$CATALOG$OBJECT(
    PARAMETER$OBJECT$TOKEN,
    TERM$OUT$MBX,
    @(7,'TERMOUT'),
    @STATUS );
/*
    Lookup the open connection for the console.
*/
CONSOLE$TOKEN = RQ$LOOKUP$OBJECT(
    NUL,
    @(7,'CONSOLE'),
    WAIT$FOREVER,
    @STATUS );
```

Figure A-7. Terminal Job (TERM.P28)
(continued)

```
/*
    Create the response mailbox to be used with BIOS system calls.
*/
BIOS$RSP$MBX = RQ$CREATE$MAILBOX(
    0,
    @STATUS );
DO WHILE TRUE;
/*
    Enter forever loop waiting for input requests.
*/
    /*
        Wait for an output message.
    */
    BUFFERT = RQ$RECEIVE$MESSAGE(
        TERM$OUT$MBX,
        WAIT$FOREVER,
        @USER$RSPMBX,
        @STATUS );
    /*
        Write the message to the console
    */
    CALL RQ$A$WRITE(
        CONSOLE$TOKEN,
        @BUFFER,
        STRLEN(@BUFFER),
        BIOS$RSP$MBX,
        @STATUS );
    /*
        Wait for the message to be output
    */
    ACTUAL = RQ$WAIT$IO(
        CONSOLE$TOKEN,
        BIOS$RSP$MBX,
        WAIT$FOREVER,
        @STATUS );
    /*
        Now delete the segment which we received at TERMOUTMBX.
    */
    CALL RQ$DELETE$SEGMENT(
        BUFFERT,
        @STATUS );
    END; /* DO WHILE 1 */

END TERMOUT_TSK;
```

Figure A-7. Terminal Job (TERM.P28)
(continued)

```
TERM_INIT_TSK:  PROCEDURE   PUBLIC;
/*
    This task is responsible for setting up the connections to
    the console and creating the tasks TERMINTSK, and TERMOUTTSK.
*/
DECLARE
    MBX,
    BIOS$RSP$MBX                TOKEN,
    DUMMY$MBX                   TOKEN,
    TERM$DEV$CON                TOKEN,
    TERM$FILE$CON               TOKEN,
    USER$T                      TOKEN,
    TERMIN$TASK$TOKEN           TOKEN,
    TERMOUT$TASK$TOKEN          TOKEN,
    IORST                       TOKEN,

    IORS BASED IORST            STRUCTURE   (
        STATUS  WORD ),
    STATUS,
    ACTUAL  )               WORD,

    BUFFER( 256 )               BYTE,

    SIGNON$MESSAGE  (*)         BYTE        DATA        'TERMINAL ONLINE --
READY FOR PROCESSING.',0AH,0DH );

/*
    Create the response mailbox to be used with BIOS system calls.
*/
BIOS$RSP$MBX = RQ$CREATE$MAILBOX(
    0,
    @STATUS );
/*
    Create a device connection the terminal.
*/
CALL RQ$APHYSICAL$ATTACH$DEVICE(
    @(2,'T0'),
    PHYSICAL$FILE$DRIVER,
    BIOS$RSP$MBX,
    @STATUS );
```

Figure A-7. Terminal Job (TERM.P28) (continued)

```
/*
    Wait for the device connection token to be returned be the BIOS.
*/
TERM$DEV$CON = RQ$RECEIVE$MESSAGE(
    BIOS$RSP$MBX,
    WAIT$FOREVER,
    @DUMMY$MBX,
    @STATUS );
/*
    Create a file connection on the terminal device.
*/
CALL RQ$A$CREATE$FILE(
    NUL,                    /* User token ignored for physical files.  */
    TERM$DEV$CON,
    @(0),              /* Subpath pointer ignored for physical files.  */
    DRAU,
    0,                 /* Granularity irrelevant.  We are not on a */
                       /* Random Access device.  */
    0,                 /* Size irrelevant.  We are not on a Random */
                       /* Access device.  */
    0,
    BIOS$RSP$MBX,
    @STATUS );
/*
    Wait for the device connection token to be returned be the BIOS.
*/
TERM$FILE$CON = RQ$RECEIVE$MESSAGE(
    BIOS$RSP$MBX,
    WAIT$FOREVER,
    @DUMMY$MBX,
    @STATUS );
/*
    Open the terminal file for both read and write.
*/
CALL RQ$A$OPEN(
    TERM$FILE$CON,
    READ$WRITE,
    SHARE$ALL,
    BIOS$RSP$MBX,
    @STATUS );
```

Figure A-7. Terminal Job (TERM.P28)
(continued)

```
/*
    Wait for the I/O result segment to be returned by the BIOS.
*/
IORST = RQ$RECEIVE$MESSAGE(
    BIOS$RSP$MBX,
    WAIT$FOREVER,
    @DUMMY$MBX,
    @STATUS );
/*
    Catalog the connection token for the open file in the local job as
    "CONSOLE".
*/
CALL RQ$CATALOG$OBJECT(
    NUL,
    TERM$FILE$CON,
    @(7,'CONSOLE'),
    @STATUS );
/*
    Create the task that will handle keyboard input.
*/
TERMIN$TASK$TOKEN = RQ$CREATE$TASK(
    139,
    @TERMIN_TSK,
    SELECTOR$OF(@STATUS),    /* Initialize the DS as we are in COMPACT */
    NIL,                     /* Let rmx assign the stack */
    400H,                    /* Stack Size */
    NO$NPX,                  /* This task does not use the NPX */
    @status );
/*
    Create the task which will process console output.
*/
TERMOUT$TASK$TOKEN = RQ$CREATE$TASK(
    139,
    @TERMOUT_TSK,
    SELECTOR$OF(@STATUS),    /* Initialize the DS as we are in COMPACT */
    NIL,                     /* Let rmx assign the stack */
    400H,                    /* Stack Size */
    NO$NPX,                  /* This task does not use the NPX */
    @status );
/*
    Delete thyself as nothing more to do.
*/
CALL RQ$DELETE$TASK(
    NUL,
    @STATUS );
```

Figure A-7. Terminal Job (TERM.P28) (continued)

```
IF STATUS <> 0 THEN
    CAUSEINTERRUPT( 3 );

END TERM_INIT_TSK;

END TERM_MOD;
```

Figure A-7. Terminal Job (TERM.P28) (continued)

```
$COMPACT DEBUG PW(79) ROM
SBX350_MOD:  DO;

$INCLUDE(/RMX286/INC/RMXPLM.EXT)

DECLARE
    NUL                 LITERALLY       'SELECTOR$OF(NIL)',
    /*
        Use MINTR0 from an iSBX 350 installed on J5
    */
    SBX350$INTR$LEVEL           LITERALLY       '075H',
    WAIT$FOREVER                LITERALLY       '0FFFFH',
    NO$WAITING                  LITERALLY       '0',
    ONE$SECOND                  LITERALLY       '0100',
    CR                          LITERALLY       '0DH',
    LF                          LITERALLY       '0AH',
    SBX350$PORT$A               LITERALLY       '0A0H',
    SBX350$PORT$B               LITERALLY       '0A2H',
    SBX350$PORT$C               LITERALLY       '0A4H',
    SBX350$CONTROL$PORT         LITERALLY       '0A6H',
    /*
        Set Port A for strobed input and Port B for Mode 0 output
    */
    SBX350$INIT$MODE            LITERALLY       '10111000B',
    ENABLE$PORT$A$INTERRUPT     LITERALLY       '00001001B';
```

Figure A-8. Interrupt Handler and Task (SBX350.P28)

```
SBX350$INT$HNDLR:  PROCEDURE    INTERRUPT  PUBLIC;

DECLARE STATUS WORD;

*/
    Let the interrupt task do all of the work.  The time for a
    context switch should help in the debounce of the interrupt switch.
*/
CALL RQ$SIGNAL$INTERRUPT( SBX350$INTR$LEVEL, @STATUS );

END     SBX350$INT$HNDLR;

SX350 TSK:  PROCEDURE   PUBLIC;

DECLARE
    SBX350MBX               TOKEN,
    PROCIOMBX               TOKEN,
    RSPMBX                  TOKEN,
    STATUS                  WORD,
    INPUT$VALUES (128)      BYTE,
    ACTUAL                  BYTE,
    NUMBER$OF$SAMPLES       BYTE,
    SBX$INIT$DATA           BYTE,
    INDEX                   BYTE;
DECLARE
    SBXINITMSG(*)           BYTE  DATA (
    CR,LF,'THE INTERRUPT TASK IS NOW INITIALIZED',CR,LF,0);

/*
    Initialize the SBX350 interface.
    Port A = Input
    Port B = Output
    Port C = Output
    Port A interrupt enabled
*/
OUTPUT(SBX350$CONTROL$PORT) = SBX350$INIT$MODE;
OUTPUT(SBX350$CONTROL$PORT) = ENABLE$PORT$A$INTERRUPT;

SBX350$MBX = RQ$CREATE$MAILBOX (
    60H,            /*  Set up for passing data not RMX objects */
    @STATUS );
```

Figure A-8. Interrupt Handler and Task (SBX350.P28) (continued)

```
CALL RQ$CATALOG$OBJECT(
    NUL,
    SBX350$MBX,
    @(9,'SBX350MBX'),
    @STATUS );

/*
    Install the Interrupt Handler.
*/
CALL RQ$SET$INTERRUPT(
    SBX350$INTR$LEVEL,
    1,
    @SBX350$INT$HNDLR,
    NUL,
    @STATUS);

PROCIOMBX = RQ$LOOKUP$OBJECT(
    NUL,
    @(9,'PROCIOMBX'),
    WAIT$FOREVER,
    @STATUS );

CALL RQ$SEND$DATA(
    PROCIOMBX,
    @SBXINITDATA,
    1,
    @STATUS );
ACTUAL = RQ$RECEIVE$DATA (
    SBX350MBX,
    @NUMBER$OFSAMPLES,
    WAIT$FOREVER,
    @STATUS );

DO WHILE 1;
/*
    Enter forever loop waiting for interrupts.
*/
    DO INDEX = 0 TO ( NUMBER$OF$SAMPLES - 1 );
        CALL RQ$WAIT$INTERRUPT(
            SBX350$INTR$LEVEL,
            @STATUS );
```

Figure A-8. Interrupt Handler and Task (SBX350.P28) (continued)

```
        /*
            Wait to debounce the interrupt switch.
        */
        CALL RQ$SLEEP(
            1,
            @STATUS);
        /*
            Get the input data.
        */
        INPUT$VALUES( INDEX ) = NOT (INPUT( SBX350$PORT$A ));
        OUTPUT( SBX350$PORT$B ) = INPUT$VALUES( INDEX );
        END; /* DO INDEX = 0 TO ( NUMBER$OF$SAMPLES - 1 ) */

    CALL RQ$SEND$DATA(
        PROCIOMBX,
        @INPUT$VALUES,
        NUMBER$OF$SAMPLES,
        @STATUS );
    ACTUAL = RQ$RECEIVE$DATA (
        SBX350MBX,
        @NUMBER$OFSAMPLES,
        NO$WAITING,
        @STATUS );

    END; /* WHILE 1 */

END SBX350_TSK;

END SBX350_MOD;
```

Figure A-8. Interrupt Handler and Task (SBX350.P28) (continued)

```
$COMPACT DEBUG PW(79) ROM

PROCIO_MOD:  DO;

$INCLUDE(/RMX286/INC/RMXPLM.EXT)

STRLEN:  PROCEDURE( STRPTR )    BYTE  EXTERNAL;
    DECLARE STRPTR  POINTER;
END STRLEN;

BTOA:  PROCEDURE( CHR, STR )    EXTERNAL;
    DECLARE
        CHR BYTE,
        STR POINTER;
END  BTOA;

ATOB:  PROCEDURE( STR ,STSP )   BYTE    EXTERNAL;
    DECLARE
        STR     POINTER,
        STSP        POINTER;
END ATOB;


DECLARE
    NUL                 LITERALLY           'SELECTOR$OF(NIL)',
    WAIT$FOREVER        LITERALLY           '0FFFFH',
    NO$WAITING          LITERALLY           '0',
    E$OK                LITERALLY           '0',
    CR                  LITERALLY           '0DH',
    LF                  LITERALLY           '0AH',
    TRUE                LITERALLY           '0FFH',
    FALSE               LITERALLY           '0';


CALC$MEAN:  PROCEDURE( COUNT, BUFFP )   BYTE  EXTERNAL;
DECLARE
    COUNT   BYTE,
    BUFFP   POINTER;
END CALC$MEAN;

CALC$MEDIAN:  PROCEDURE( COUNT, BUFFP ) BYTE    EXTERNAL;
DECLARE
    COUNT   BYTE,
    BUFFP   POINTER;
END CALC$MEDIAN;
```

Figure A-9. Process Task (PROCIO.P28)

```
CALC$MODE:  PROCEDURE( COUNT, BUFFP ) BYTE  EXTERNAL;
DECLARE
    COUNT   BYTE,
    BUFFP   POINTER;
END CALC$MODE;

PROCIO_TSK:  PROCEDURE  PUBLIC;
DECLARE
    TERMINMBX                          TOKEN,
    TERMOUTMBX                         TOKEN,
    SBX350$MBX                         TOKEN,
    PROC$IO$MBX                        TOKEN,
    PROC$IO$RSP$MBX                    TOKEN,
    DUMMY$MBX                          TOKEN,
    BUFFERT                            TOKEN,
    STATUS                             WORD,
    ACTUAL                             WORD,
    NUMBER$OF$SAMPLES                  BYTE,
    LAST$SAMPLE                        BYTE,
    MEAN                               BYTE,
    MEDIAN                             BYTE,
    MODE                               BYTE,
    WAITING$FOR$KEYBOARD               BYTE,
    VALID                              BYTE,
    DATA$BUFFER (100H)                 BYTE,
    BUFFER  BASED  BUFFERT (256)       BYTE;

DECLARE
    INITMSG (*)     BYTE    DATA(
        CR,LF,' INPUT INITIALIZATION COMPLETE ',CR,LF,0 ),
    MEAN$MSG (*)        BYTE        DATA    (
    CR,LF,' THE MEAN INPUT VALUE WAS   '),
    MEDIAN$MSG (*)      BYTE        DATA    (
    CR,LF,'THE MEDIAN INPUT VALUE WAS  '),
    MODE$MSG (*)        BYTE        DATA    (
    CR,LF,'THE MODE INPUT VALUE WAS    '),
    ENTRY$MSG (*)       BYTE        DATA    (
        CR,LF,'PLEASE ENTER A DECIMAL VALUE BETWEEN 1 AND 128 <CR> ',0 ),
    SAMPLE$MSG (*)      BYTE        DATA    (
        CR,LF,'THE NUMBER OF SAMPLES TAKEN WAS   ');
```

Figure A-9. Process Task (PROCIO.P28)
(continued)

```
/*
    Lookup the mailboxes for terminal I/O and the SBX350 task.
*/
TERM$IN$MBX = RQ$LOOKUP$OBJECT(
    NUL,
    @(6,'TERMIN'),
    WAIT$FOREVER,
    @STATUS );
TERM$OUT$MBX = RQ$LOOKUP$OBJECT(
    NUL,
    @(7,'TERMOUT'),
    WAIT$FOREVER,
    @STATUS );
SBX350$MBX = RQ$LOOKUP$OBJECT(
    NUL,
    @(9,'SBX350MBX'),
    WAIT$FOREVER,
    @STATUS );
/*
    Setup a mailbox where the PPI task can send data for processing.
*/
PROCIO$MBX = RQ$CREATE$MAILBOX (
    60H,        /*  Set up for passing data not RMX objects */
    @STATUS );
/*
    Catalog the mailbox so the PPI task can find it.
*/
CALL RQ$CATALOG$OBJECT(
    NUL,
    PROCIO$MBX,
    @(9,'PROCIOMBX'),
    @STATUS );
/*
    Wait for data from PPI task initialization before proceeding.
*/
ACTUAL = RQ$RECEIVE$DATA(
    PROCIO$MBX,
    @DATA$BUFFER,
    WAIT$FOREVER,
    @STATUS );
```

Figure A-9. Process Task (PROCIO.P28) (continued)

```
/*
    Create a mailbox for use when requesting  keyboard input
*/
PROCIO$RSP$MBX = RQ$CREATE$MAILBOX (
    0,          /*  Set up for passing RMX objects */
    @STATUS );
/*
    Wait for a response from the operator before proceeding.
*/
VALID = FALSE;
DO WHILE NOT VALID;
/*
    Wait for initialization data from the operator before proceeding.
*/
    /*
        send a message requesting operator input
    */
    BUFFERT = RQ$CREATE$SEGMENT(
        SIZE(BUFFER),
        @STATUS );
    CALL MOVB(
        @ENTRY$MSG,
        @BUFFER,
        SIZE( ENTRY$MSG ) );
    CALL RQ$SEND$MESSAGE(
        TERM$OUT$MBX,
        BUFFERT,
        NUL,
        @STATUS );
    /*
        Now check for a response.
    */
    BUFFERT = RQ$CREATE$SEGMENT(
        SIZE(BUFFER),
        @STATUS );
    BUFFER(0) = 80H;
    CALL RQ$SEND$MESSAGE(
        TERM$IN$MBX,
        BUFFERT,
        PROC$IO$RSP$MBX,
        @STATUS );
    BUFFERT = RQ$RECEIVE$MESSAGE (
        PROC$IO$RSP$MBX,
        WAIT$FOREVER,
        @DUMMY$MBX,
        @STATUS );
```

Figure A-9. Process Task (PROCIO.P28) (continued)

```
    /*
        Validate the input parameter and transform it from ASCII
    */
    NUMBER$OF$SAMPLES  = ATOB(
        @BUFFER,
        @STATUS );
    IF ( STATUS = 0 )
    AND (NUMBER$OF$SAMPLES >0 )
    AND ( NUMBER$OF$SAMPLES <=128 ) THEN
        VALID = TRUE;
    /*
        Delete the segment received at PROCIO$RSP$MBX.
    */
    CALL RQ$DELETE$SEGMENT(
        BUFFERT,
        @STATUS );
    END; /* DO WHILE NOT VALID */
    LAST$SAMPLE = NUMBER$OFSAMPLES;

/*
    Now let the PPI task proceed
*/

CALL RQ$SEND$DATA(
    SBX350$MBX,
    @NUMBER$OF$SAMPLES,
    1,
    @STATUS);
/*
    Set a flag noting the no requests are pending for keyboard input.
*/
WAITING$FOR$KEYBOARD = FALSE;
DO WHILE 1;
/*
    Enter main loop for processing digital I/O.
*/

    /*
        Wait for data from PPI task.
    */
    ACTUAL = RQ$RECEIVE$DATA(
        PROCIO$MBX,
        @DATA$BUFFER,
        WAIT$FOREVER,
        @STATUS );
```

Figure A-9. Process Task (PROCIO.P28)
(continued)

```
/*
    Now format the data and send it to the terminal output task.
*/

/*
    Inform the operator of the number of samples taken.
*/
BUFFERT = RQ$CREATE$SEGMENT(
    SIZE(BUFFER),
    @STATUS );
CALL MOVB(
    @SAMPLE$MSG,
    @BUFFER,
    SIZE( SAMPLE$MSG));
CALL BTOA(
    ACTUAL,
    @BUFFER( SIZE( SAMPLE$MSG)));
CALL RQ$SEND$MESSAGE(
    TERM$OUT$MBX,
    BUFFERT,
    NUL,
    @STATUS );
/*
    Calculate the mean input value send it to the operator
    as part of a zero terminated string..
*/
MEAN = CALC$MEAN(
    ACTUAL,
    @DATA$BUFFER );
BUFFERT = RQ$CREATE$SEGMENT(
    SIZE(BUFFER),
    @STATUS );
CALL MOVB(
    @MEAN$MSG,
    @BUFFER,
    SIZE( MEAN$MSG));
CALL BTOA( MEAN,
    @BUFFER( SIZE( MEAN$MSG)));
CALL RQ$SEND$MESSAGE(
    TERM$OUT$MBX,
    BUFFERT,
    NUL,
    @STATUS );
```

Figure A-9. Process Task (PROCIO.P28) (continued)

```
/*
    Calculate the median input value send it to the operator
    as part of a zero terminated string..
*/
MEDIAN = CALC$MEDIAN(
    ACTUAL,
    @DATA$BUFFER );
BUFFERT = RQ$CREATE$SEGMENT(
    SIZE(BUFFER),
    @STATUS );
CALL MOVB(
    @MEDIAN$MSG,
    @BUFFER,
    SIZE( MEDIAN$MSG));
CALL BTOA(
    MEDIAN,
    @BUFFER( SIZE( MEDIAN$MSG)));
CALL RQ$SEND$MESSAGE(
    TERM$OUT$MBX,
    BUFFERT,
    NUL,
    @STATUS );
/*
    Calculate the mode input value send it to the operator
    as part of a zero terminated string..
*/
MODE = CALC$MODE(
    ACTUAL,
    @DATA$BUFFER );
BUFFERT = RQ$CREATE$SEGMENT(
    SIZE(BUFFER),
    @STATUS );
CALL MOVB(
    @MODE$MSG,
    @BUFFER,
    SIZE( MODE$MSG));
CALL BTOA(
    MODE,
    @BUFFER( SIZE( MODE$MSG)));
CALL RQ$SEND$MESSAGE(
    TERM$OUT$MBX,
    BUFFERT,
    NUL,
    @STATUS );
```

Figure A-9. Process Task (PROCIO.P28) (continued)

```
/*
    send a message requesting operator input
*/
BUFFERT = RQ$CREATE$SEGMENT(
    SIZE(BUFFER),
    @STATUS );
CALL MOVB(
    @ENTRY$MSG,
    @BUFFER,
    SIZE( ENTRY$MSG));
CALL RQ$SEND$MESSAGE(
    TERM$OUT$MBX,
    BUFFERT,
    NUL,
    @STATUS );

IF WAITING$FOR$KEYBOARD <> TRUE THEN DO;
/*
    If no input request message is pending then send one.
*/
    BUFFERT = RQ$CREATE$SEGMENT(
        SIZE( BUFFER),
        @STATUS);
    BUFFER(0) = 80H;
    CALL RQ$SEND$MESSAGE(
        TERM$IN$MBX,
        BUFFERT,
        PROC$IO$RSP$MBX,
        @STATUS);
    WAITING$FOR$KEYBOARD = TRUE;
    END;
/*
    Check for a response from the operator before proceeding.
*/
BUFFERT = RQ$RECEIVE$MESSAGE (
    PROC$IO$RSP$MBX,
    NO$WAITING,
    @DUMMY$MBX,
    @STATUS );
IF STATUS = 0 THEN DO;
```

Figure A-9. Process Task (PROCIO.P28)
(continued)

```
    /*
        Validate the input parameter and transform it from ASCII.
    */
        NUMBER$OF$SAMPLES  = ATOB (
            @BUFFER,
            @STATUS );
        IF (NUMBER$OF$SAMPLES >0 )
        AND ( NUMBER$OF$SAMPLES <=128 )
        AND (STATUS = 0 ) THEN DO;
        /*
            New sample size requested.  Inform the interrupt task.
        */
            CALL RQ$SEND$DATA(
                SBX350$MBX,
                @NUMBER$OF$SAMPLES,
                1,
                @STATUS);
            LAST$SAMPLE = NUMBER$OF$SAMPLES;
            END;
        /*
            Delete the segment we received from the keyboard.
        */
        CALL RQ$DELETE$SEGMENT(
            BUFFER$T,
            @STATUS);
        /*
            Clear the flag showing that a keyboard input request is pending
        */
        WAITING$FOR$KEYBOARD = FALSE;
        END; /* IF STATUS = 0 */

END; /* DO WHILE 1 */

END PROCIO_TSK;

END PROCIO_MOD;
```

Figure A-9. Process Task (PROCIO.P28)
(continued)

```
$COMPACT DEBUG PW(79) ROM OPTIMIZE(0)

UTILITY_MOD:  DO;

$INCLUDE(/RMX286/INC/RMXPLM.EXT)

DECLARE
    NUL                          LITERALLY       'SELECTOR$OF(NIL)',
    WAIT$FOREVER                 LITERALLY       '0FFFFH',
    TRUE                         LITERALLY       '1',
    FALSE                        LITERALLY       '0',
    MAX$STRING                   LITERALLY       '255';

STRLEN:  PROCEDURE( STRPTR )     BYTE    PUBLIC;

DECLARE
    STRPTR                       POINTER,
    STRING BASED STRPTR(1)       BYTE,
    INDEX                        BYTE;

INDEX = 0;
DO WHILE ( STRING(INDEX) <> 0 ) AND ( INDEX < MAXSTRING );
    INDEX = INDEX + 1;
    END;
IF STRING(INDEX) = 0 THEN
    RETURN INDEX;
ELSE
    RETURN 0;
END STRLEN;

BTOA:  PROCEDURE( CHR, STRING$P )   PUBLIC;
    DECLARE
        STRING$P                    POINTER,
        CHR                         BYTE;

    DECLARE (
        QUOTIENT,
        REMAINDER,
        COUNT,
        MORE$DATA,
        INDEX)                      BYTE,
        STRING BASED STRING$P(80)   BYTE,
        DIGIT(80)                   BYTE;
```

Figure A-10. Utility Procedures (UTILS.P28)

```
    INDEX,MORE$DATA = 0;

    REMAINDER = CHR;

    IF CHR = 0 THEN DO;
        DIGIT(0) = '0';
        DIGIT(1) = 0;
        INDEX = 1;
        END;
    ELSE DO WHILE REMAINDER > 0;
        DIGIT( INDEX ) = ( REMAINDER MOD 10 ) + '0';
        REMAINDER = REMAINDER - ( REMAINDER MOD 10 );
        REMAINDER = REMAINDER / 10;
        INDEX = INDEX + 1;
        DIGIT( INDEX ) = 0;
        END;
    /*
        This string must be reversed to have the most significant
        digit output first at the terminal.
    */
    COUNT = 0;
    DO WHILE INDEX > 0;
        STRING( COUNT ) = DIGIT( INDEX - 1 );
        COUNT = COUNT + 1;
        INDEX = INDEX - 1;
        END;
    STRING( COUNT ) = 0;

END     BTOA;

ATOB:  PROCEDURE( STR$P, STS$P )    BYTE      PUBLIC;
/*
    This procedure is totally wrong!
*/
    DECLARE
        STR$P                   POINTER,
        STS$P                   POINTER,
        CHARS BASED STR$P(1)    BYTE,
        STATUS BASED STS$P      WORD,
        VALUE                   BYTE,
        INDEX                   BYTE,
        I                       BYTE,
        MULTIPLIER )            BYTE;
```

Figure A-10. Utility Procedures (UTILS.P28) (continued)

```
    INDEX, VALUE = 0;
    MULTIPLIER = 1;
    STATUS = 0;
    DO WHILE ( CHARS( INDEX ) >= '0')
        AND ( CHARS( INDEX ) <= '9' );
        INDEX = INDEX + 1;
        END;
    IF INDEX = 0 THEN DO;
        STATUS = 0FFFFH;
        RETURN 0;
        END;
    I = INDEX -1;
    DO WHILE I < INDEX;
        VALUE = VALUE + (( CHARS( I ) - '0' ) * MULTIPLIER);
        MULTIPLIER = MULTIPLIER * 10;
        I = I - 1;
        END;

    RETURN  VALUE;
END ATOB;

CALC$MEAN:  PROCEDURE( COUNT, BUFFP )   BYTE  PUBLIC;
DECLARE
    COUNT                       BYTE,
    BUFFP                       POINTER,
    BUFF  BASED  BUFFP(1)       BYTE,
    INDEX                       BYTE,
    TOTAL                       WORD;

TOTAL = 0;
DO INDEX = 1 TO COUNT;
    TOTAL = TOTAL + BUFF( INDEX - 1 );
    END;
RETURN ( TOTAL / COUNT ) ;

END CALC$MEAN;

CALC$MEDIAN:  PROCEDURE( COUNT, BUFFP ) BYTE    PUBLIC;
DECLARE
    COUNT                       BYTE,
    BUFFP                       POINTER,
    BUFF  BASED  BUFFP(1)       BYTE,
    VALUES(100H)                BYTE,
    TOTAL                       BYTE,
    INDEX                       BYTE;
```

Figure A-10. Utility Procedures (UTILS.P28) (continued)

```
CALL SETB( 0, @VALUES, SIZE( VALUES ) );
DO INDEX = 1 TO COUNT;
    VALUES( BUFF( INDEX - 1 ) ) = VALUES( BUFF( INDEX - 1 ) ) + 1;
    END;
TOTAL = 0;
INDEX = 0;
DO WHILE TOTAL <= ( COUNT / 2);
    TOTAL = TOTAL + VALUES( INDEX );
    INDEX = INDEX + 1;
    END;
RETURN INDEX -1;

END CALC$MEDIAN;

CALC$MODE:  PROCEDURE( COUNT, BUFFP ) BYTE  PUBLIC;
DECLARE
    COUNT               BYTE,
    BUFFP               POINTER,
    BUFF  BASED  BUFFP(1)   BYTE,
    VALUES(100H)            BYTE,
    (
    MAX,
    INDEX ) BYTE;

CALL SETB( 0, @VALUES, SIZE( VALUES ));
DO INDEX = 1 TO COUNT;
    VALUES( BUFF( INDEX - 1 )) = VALUES( BUFF( INDEX - 1 )) + 1;
    END;
MAX = 0;
DO INDEX = 0 TO LAST(VALUES);
    IF VALUES( INDEX ) > VALUES( MAX ) THEN
        MAX = INDEX;
    END;
    RETURN MAX;

END CALC$MODE;

END UTILITY_MOD;
```

Figure A-10. Utility Procedures (UTILS.P28)
(continued)

A.7.2 Compiling and Binding the Code

The following commands are used to compile the code for this example:

These PLM286 commands do not include controls for selecting the model of segmentation (SMALL, COMPACT, MEDIUM, or LARGE) because the $COMPACT control was already included in the source files. The ROM control is also included, to place the constants in the data segment.

The compiler places the generated object code into the files INITTSK.OBJ, SBX350.OBJ, TERM.OBJ, PROCIO.OBJ, and UTILS.OBJ.

After compiling, the object files must be bound with the appropriate iRMX II interface libraries. Because the code will be configured into the system with the Interactive Configuration Utility, the BND286 command required is different than the ones used for the previous examples. The BND286 command for this example must include the NOLOAD and NOPUBLICS EXCEPT controls.

The NOLOAD control is required because the SUBMIT file created by the ICU will invoke BLD286 to further process the resulting module. The NOPUBLICS EXCEPT control is required to limit the public symbols in the output module. The only public symbols in the output module are the entry point of the initial task and the dummy variable identifying the data segment of that task. If all public symbols were allowed, the names of the iRMX II interface procedures would conflict with the names of the call gates used inside the operating system.

Even though the NOPUBLICS EXCEPT control is required, including that control can make debugging somewhat of a chore because the MAP286 utility will not be able to generate much valuable information. To correct that problem, you can use a three-step bind operation that produces one output file for use by MAP286 and another output file that will be used as input to the ICU.

To implement this solution, first bind the object code with the NOLOAD and DEBUG controls and place the result in a temporary file. Then use the temporary file as input to BND286 and include the LOAD option. This will cause the error

```
NO START ADDRESS FOUND IN OUTPUT MODULE
```

This error can be ignored. The object module produced from this invocation of BND286 can be used as input to MAP286 to generate a map of the first-level job. Finally, run the temporary file through BND286 again, specifying the NOLOAD and NOPUBLICS EXCEPT controls. The output of this invocation will be combined during configuration with the rest of the operating system.

The following commands perform the operations just mentioned. They create both a usable map file and the object module that will be combined with the operating system.

A.7.3 Configuring the First-Level Job

To add the first-level job to the operating system, you must use the ICU and build a configuration containing that job. The easiest way to do this is to start with an existing configuration file and modify it to include the new first-level job. You can use any configuration file that includes the SDB, the Basic I/O System, and supports a device with the name T0. This example assumes that you start with the configuration file 28612.DEF as supplied with the operating system.

When you run the ICU, make sure to examine the following screens:

Memory for System (MEMS) screen.	In many cases, when you add a first-level job to the operating system, you must allocate more memory for the operating system. In this instance, you won't need to allocate more memory because you will be eliminating some of the layers of the operating system from your configuration (only the Nucleus, SDB, and BIOS are needed). In fact, you'll probably have more than enough memory allocated for operating system code. If you were building an actual production system, you could find out exactly how much you need by running the ICU twice. The first time through, allocate a generous amount of system memory. After you run the SUBMIT file generated by the ICU, examine the .MP2 file produced by BLD286 and find out how much memory was actually used. Then go back and modify the MEMS screen accordingly, and generate the system again. For this example, just leave the MEMS screen set to its default value.
Sub-systems (SUB) screen.	This example requires just the services of the Nucleus, System Debugger, and Basic I/O System. Therefore, specify NO for the UDI (UDI = NO) and YES for the Basic I/O System (BIO = YES).
Device Drivers.	There are no changes required to the BIOS screens, but only the 8274 driver is required for this example. To remove the other drivers, invoke their driver screens and enter the value ^D to delete them from the configuration.

User Jobs (USERJ) screen.	Because you are adding a first-level job, you must fill out the User Jobs screen to describe that job. The fields on this screen are similar to the parameters in the RQE$CREATE$JOB system call. Figure A-11 shows the filled-out User Jobs screen. Notice that the Task Start Address (TSA) and Public Variable Name (VAR) fields are set to the public names that remain in the first-level job after running BND286 (DUMMYVARIBLE and INIT_TSK).
User Modules (USERM) screen.	On this screen, you must indicate the pathname of the object module containing your first-level job. This is the file produced by BND286 earlier (EXAMPLE.LNK).

After you make these modifications with the ICU, generate a new system and run the SUBMIT file produced by the ICU. You can invoke your system by using the Bootstrap Loader to load the operating system file produced by the ICU's SUBMIT file.

```
(USERJ)       User Jobs

(NAM)   Job Name [0-14 characters]                          EXAMPLE
(ODS)   Object directory Size [0-3840]                         40
(PMI)   Pool Minimum [20H - 0FFFFFH]                         0FFFH
(PMA)   Pool Maximum [20H - 0FFFFFH]                       0FFFFFH
(MOB)   Maximum Objects [1 - 0FFFFH]                        0FFFFH
(MTK)   Maximum Tasks [1 - 0FFFFH]                          0FFFFH
(MPR)   Maximum Priority [0 - 255]                             0
(EHS)   Exception Handler Entry Point [1-31 chars]

(EM)    Exception Mode [Never/Prog/Environ/All]              ALL
(PV)    Parameter Validation [Yes/No]                        YES
(TP)    Task Priority [0-255]                                139
(TSA)   Task Entry Point [1-31 chars]
                                                          INIT_TSK
(VAR)   Public Variable Name [0-31 chars]
                                                  DUMMYDATAVARIBLE
(SSA)   Stack Segment Address [SS:SP]                 0000:0000H
(SSI)   Stack Size [0-0FFFFH]   0300H
(NPX)   Numeric Processor Extension Used [Yes/No]            NO
```

Figure A-11. User Jobs Screen

$SLEEP A-32, 35

A

AGETCONNECTION$STATUS A-6
AGETFILE$STATUS A-6
A$OPEN A-6, 61
A$PHYSICAL$ATTACH$DEVICE A-6
A$READ A-21, 57
A$SEEK A-6, 21
A$SPECIAL A-6, 7, 8
A$WRITE A-21, 23, 59
Assembly code
    Parameter passing 2-3
    Using iRMX® system calls 2-1

B

Based variables 5-2
Bind sequence 2-12, A-23, 37, 41, 48, 81
Binding code to interface libraries 2-11
BIOS.EXT 2-5
BND286 2-11, 12, A-37, 41, 48, 81
Buffer Pools
    Creating 5-3, A-16
    Overview A-3
    Passing buffers A-15
    Releasing buffers A-17, 19
    Requesting buffers A-18
    Using 5-3, A-15, 18

C

C code
    Parameter passing 2-4
    Using iRMX® system calls 2-4
CGETOUTPUT$CONNECTION A-6
CGETOUTPUT$PATHNAME A-6
C$SEND$CO$RESPONSE A-32, 35
C$SEND$EO$RESPONSE A-20
CATALOG$OBJECT A-11, 16, 29, 56, 58, 62, 65, 69

Cataloging objects A-2, 10
Coding iRMX® system calls 2-1
Communication A-3
  Jobs 3-1
  Tasks A-12, 22
Compact segmentation model 1-2
Compiler controls
  RAM 1-2, 4
  ROM 1-2, 4
Compiling code A-23, 37, 41, 48, 80
Configuring a job into the operating system A-81
Constant locations 1-4
Control-C handler A-38
Conventions iv
Converting iRMX® I applications 5-1
CREATE$BUFFER$POOL A-17, 18
CREATEIOJOB A-6
CREATE$JOB A-5
CREATE$MAILBOX A-14, 29, 34, 56, 58, 59, 60, 64, 69, 70
CREATE$SEGMENT A-17, 18, 34, 70, 72, 73, 74
CREATE$TASK A-9, 11, 30, 53, 62
Creating tasks A-2, 9

D

Data Acquisition job A-48
Data passing between jobs 3-2
  SEND$DATA and RECEIVE$DATA 3-6
Dynamic stack allocations 1-3

E

E$CREATE$IO$JOB A-6
E$CREATE$JOB A-54
EIOS.EXT 2-5
Example
  Binding code A-23, 37, 41, 48, 81
  BND286 2-12
  Buffer Pools A-15
  Cataloging objects A-10
  Clearing the screen A-20
  Compiling code A-23, 37, 41, 48, 80

Example (cont.)
Concepts A-1
Configuring a job into the operating system A-83
Control-C handler A-38
Data acquisition A-48
Execution of a job A-24, 38, 41, 48
First-level job A-48
FORTRAN-286 string conversion 2-10
In-line exception processing A-3
Include files 2-5, 6, 7
Inter-task communication A-12, 22, 26
Interrupt handler A-43, 49, 63
Job initialization A-52
Literal files A-5
Programming concepts A-1
Programs A-1
Pushing parameters onto the stack 2-3
Response pointer A-12
Screen I/O A-20, 35, 55
Simultaneous I/O A-22
System calls from assembly source code 2-3
Task creation A-9
Terminal attributes A-7
Exception handlers 1-3, 5-4, A-4
Execution of a job A-24, 38, 41, 48
EXITIOJOB A-6, 27, 30
External procedures 1-2, 2-4, 11

F

File connection restrictions 3-3, 6
FORTRAN-286 code
Include file 2-7
Parameter passing 2-8
Restrictions 2-8
String conversion example 2-10
Strings 2-10
Using iRMX® system calls 2-8

G

General protection error 3-1, 6
GET$EXCEPTION$HANDLER A-4, 5
GET$LOGICAL$DEVICE$STATUS A-6
GET$PRIORITY A-9, 11, 29
GET$TASK$TOKENS A-5, 29, 53, 56, 58
GET$TYPE A-5, 32
Getting terminal attributes A-7

H

HI.EXT 2-5

I

I/O Result Segment (IORS) A-21, 22, 62
Improving performance (see optimization) 5-1
In-line exception processing A-2, 3
Include files
    BIOS.EXT 2-5
    Description 2-5, A-36
    EIOS.EXT 2-5
    Example of use 2-5, 6
    Example use of 2-7
    FORTRAN-286 file 2-7
    HI.EXT 2-5
    LOADER.EXT 2-5
    Location of 2-5
    NUCLUS.EXT 2-5
    PASCAL-286 file 2-6
    RMXFTN.EXT 2-5, 7, 8
    RMXPAS.EXT 2-5, 6, 7
    RMXPLM.EXT 2-5
    Types 2-5
    UDI.EXT 2-5
Initialization A-52
Inter-task communication A-3, 12, 26
Interface libraries
    BND286 restrictions 2-12
    Choosing for use 2-11
    Function 2-11
    RMXIFC.LIB 2-11, 12
    UDIIFC.LIB 2-11, 12
Interrupt handlers 4-1, 3, A-43, 49, 63
Interrupts 4-1

L

Large applications 1-2, 3
Literal files A-2, 5
LOADER.EXT 2-5
Loading the stack 2-3
LOGICAL$ATTACH$DEVICE A-6
LOOKUP$OBJECT A-4, 31, 34, 36, 56, 58, 65, 69

M

Mailbox 3-6, A-27, 49
Mailbox optimization 5-3
Manual overview iv
Maskable interrupts 4-1
Medium segmentation models 1-2
Multitasking job A-48

N

Nonmaskable interrupts 4-1
NUCLUS.EXT 2-5

O

Object cataloging A-2, 10
Object directory of the root job 3-5
Object passing between jobs 3-3
Optimization
    Based variables 5-2
    Compiler controls 5-2
    Mailboxes 5-3
    Nucleus 5-3
    Overflow queues, mailboxes 5-3
    Segmentation model 5-2
    Sequential I/O 5-3
    Based variables 5- 2

P

Parameter passing
    Assembly code 2-3
    Between tasks 3-6
    C code 2-4
    FORTRAN-286 2-8

PASCAL-286 code
Include file 2-6
Restrictions 2-6
Stack allocation 2-7
Passing buffers between tasks A-15
Passing data passing between jobs
BIOS 3-3
EIOS 3-3
Segments 3-2
SEND$DATA and RECEIVE$DATA 3-2
Stream files 3-3
UDI 3-3
Passing objects between jobs
Guidelines 3-7
Mailboxes 3-6, 7
Object directories 3-4, 7
Overview 3-3
Parameters 3-6, 7
Performance 1-1, 2-2, 6, 5-1, 2
PL/M-286 code
Based variables 5-2
Using iRMX® system calls 2-1
Processing exceptions A-3
Program conversion 5-1
Programming examples A-1
Public procedures 1-2, 2-2

R

RAM compiler control 1-2, 4
Reader level iii, 1-1, 2-1, 3-1, 4-1
RECEIVE$MESSAGE A-7, 8, 13, 14, 15, 23, 32, 35, 56, 59, 61, 62, 70, 74
RECEIVE$UNITS A-12, 15
RELEASE$BUFFER A-14, 17, 18, 19
Releasing buffers A-17, 19
REQUEST$BUFFER A-12, 18, 22
Requesting buffers A-18
RESET$INTERRUPT A-44, 45
Response pointer A-3, 12
Restrictions
BND286 2-12
Compact segmentation model 1-3
Connection objects 3-3, 6
FORTRAN-286 code 2-8

Restrictions (cont.)
Large segmentation model 1-3
Medium segmentation model 1-3
PASCAL-286 code 2-6
Passing data between jobs 3-2, 3
Passing objects between jobs 3-6
Small segmentation model 1-3
Stack size 4-2
RESUME$TASK A-36
RMXFTN.EXT 2-5, 7, 8
RMXIFC.LIB 2-11, 12
RMXPAS.EXT 2-5, 6, 7
RMXPLM.EXT 2-5
ROM compiler control 1-2, 4
Root job object directory 3-5

S

SGETCONNECTION$STATUS A-6
SGETFILE$STATUS A-6
S$OPEN A-4, 6, 21
S$SEEK A-6
S$SPECIAL A-6, 7, 8
Screen I/O A-3, 20, 35, 55
Segment registers 1-1, 5-1, 2
Segmentation model
Assembly language calling conventions 2-2
Choosing the size 1-3, 2-2, 3
Compact 1-2, 2-2, 3
Default 5-1
Interface libraries 2-11
Large 1-2, 2-2, 5-1
Medium 1-2, 2-2
Small 1-2
Semaphores
Cataloging 3-4
Creations 3-4
Getting units from 3-5, A-15
Looking up 3-5
Use in synchronization 3-4
SEND$MESSAGE A-12, 13, 14, 15, 18, 22, 33, 35, 36, 57, 70, 72, 73, 74
SEND$UNITS A-14, 15, 19
SET$EXCEPTION$HANDLER A-4, 5
SET$INTERRUPT A-44, 47, 65
Setting terminal attributes A-7

SIGNAL$INTERRUPT A-43, 46, 50, 64
Simultaneous I/O A-3, 22
SLEEP A-47, 66
Small applications 1-2, 3
Small segmentation model 1-2
Stack
  Allocation in PASCAL-286 code 2-7
  Computing size using the arithmetic technique 4-3, 4
  Computing size using the empirical technique 4-4
  Interrupt requirements 4-3
  Overflow 4-1, 2
  Recursive code 4-2
  Size for created tasks and jobs 4-2
  Size for loaded or invoked tasks 2-12, 4-2, A-37
  Size limitation for interrupt handlers 4-1, 3
  System call requirements 4-3, 4
Stream file 3-3
SUSPEND$TASK A-30, 32, 36
Synchronization
  Tasks in different jobs 3-4
  Using semaphores 3-4

T

Task creation A-2, 9
Terminal attributes A-2, 7
Terminal I/O A-20, 35, 55

U

UDI system calls 2-11
UDI.EXT 2-5
UDIIFC.LIB 2-11, 12
Using iRMX® system calls 2-1

W

WAIT$INTERRUPT A-44, 47, 65
WAIT$IO A-21, 22, 23, 57, 59
Writing code 2-1

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
Flower-Hill Shin-machi
1-23-9, Shinmachi
Setagaya-ku, Tokyo 15

NETHERLANDS
Intel Semiconductor (Netherland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor G.N.B.H.
Seidlestrasse 27
D-8000 Munchen