intel

iRMX® Bootstrap Loader Reference Manual

Order Number: 462921-001

intel®

iRMX®
Bootstrap Loader
Reference Manual

Order Number: 462921-001

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 after 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	Plug-A-Bubble
BITBUS	i_m	iRMX	PROMPT
COMMputer	iMDDX	iSBC	Promware
CREDIT	iMMX	iSBX	QUEST
Data Pipeline	Insite	iSDM	QueX
Genius	int_el	iSSB	Ripplemode
$\hat{i}$	Intel376	iSXM	RMX/80
i	Intel386	Library Manager	RUPI
I^2ICE	int_elBOS	MCS	Seamless
ICE	Intelevision	Megachassis	SLD
iCEL	$int_eligent$ Identifier	MICROMAINFRAME	UPI
iCS	$int_eligent$ Programming	MULTIBUS	VLSiCEL
iDBP	Intellec	MULTICHANNEL	376
iDIS	Intellink	MULTIMODULE	386
	iOSP	OpenNET	386SX
	iPDS	ONCE	
	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, PC/XT, and PC/AT are registered trademarks of International Business Machines. Soft-Scope is a registered trademark of Concurrent Sciences.

REV.	REVISION HISTORY	DATE
-001	Original Issue.	03/89

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® Operating Systems 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.
- ASM source code files.

MANUAL OVERVIEW

The descriptions in this manual apply to both the iRMX I and the iRMX II Operating Systems. Differences, where they exist, are highlighted in the text.

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.

Appendix C This appendix describes use of the D-MON386 monitor in the Bootstrap process.

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 blue text
```

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

- The term "iRMX II" refers to the iRMX II (iRMX 286) Operating System.
- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- The terms "iRMX" or "iRMX Operating System" when used by themselves, refer to both iRMX I and II, that is, the text applies equally to both operating systems.
- All numbers, unless otherwise stated, are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

CONTENTS

Chapter 1. Overview of Bootstrap Loader Operations

Chapter 2. Using the Bootstrap Loader

Chapter 3. Configuring the First Stage

Chapter 3. Configuring the First Stage (continued)

Chapter 4. Configuring the Third Stage

Chapter 5. Writing a Custom First-Stage Driver

Chapter 6. Writing a Custom Third-Stage Driver

Chapter 7. Error Handling

Appendix A. Automatic Boot Device Recognition

Appendix B. PROMming the Bootstrap Loader and the iSDM™ Monitor

Appendix C. D-MON386 Monitor

Index

Tables

Figures

OVERVIEW OF BOOTSTRAP LOADER OPERATIONS 1

1.1 INTRODUCTION

The iRMX® Bootstrap Loader is a program that loads your custom version of the iRMX Operating System, known as an application system, into RAM from secondary storage so 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. Since it's not dependent on any particular operating system, the iRMX Bootstrap Loader can load many different operating systems.

The Bootstrap Loader eliminates the need to place complete applications into PROM devices. Instead, you can place the Bootstrap Loader--a 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 a series of stages and is necessary for loading iRMX applications.

The first stage is located 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 300 Series Microcomputers are delivered with the first stage of the iRMX Bootstrap Loader and the iSDM™ monitor already placed in PROM devices. If you are building your own MULTIBUS® I microcomputer systems, you can use the information in this manual to configure a first stage and place it into PROM devices.

Intel System 520 Series Microcomputers are delivered with firmware containing a different Bootstrap Loader than the one described in this manual. None of the information contained in this manual applies to the System 520's Bootstrap Loader. That loader, the MULTIBUS II Systems Architecture (MSA) Bootstrap Loader, is described in the *MSA Bootstrap Loading of the System 520* Manual.

The second stage is on track 0 of every iRMX-formatted named volume and is necessary for leading iRMX I and iRMX II applications. 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. Then it loads a file from the same volume and passes control to it. This second stage does not function in the MSA environment. Refer to the *MSA Bootstrap Loading of the System 520* Manual for more information.

The contents of this load file depend on the type 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 only for loading iRMX II applications, because these applications require the 80286/386™ 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 is 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 accomplished without a device driver. The device driver is a small program that provides the interface between the Bootstrap Loader and a device controller. When you configure the Bootstrap Loader (a task that is independent of iRMX Operating System configuration), you specify the device drivers that the Bootstrap Loader requires. During configuration, these device drivers, which are distinct from the drivers needed by the application system, are linked to the Bootstrap Loader.

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. 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 300 Series Microcomputers are delivered with the Bootstrap Loader's first stage, the iSDM monitor, and the System Confidence Test (SCT) in the PROM devices.

When the first stage begins running, it first identifies the bootstrap device and the name of the file to boot. It can identify the device from a command line you enter at the monitor, or it can use default characteristics established when the first stage was configured. The Bootstrap Loader then calls its internal device driver for the device, to initialize the device and read the first portion of the second stage into memory. This occurs for all devices except when the device is located in a remote system (for example, over a Local Area Network). After calling the internal device driver, the first stage passes control to the second stage. When the bootstrap device is a remote device, the remote system does not load the second stage.

Debugging the First 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 operate in real address mode.

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 will find it useful to load the first stage into the target system's RAM using a development system iSDM loader. Then you can activate the first stage under iSDM control from the development system. After activating the first stage, you can then debug driver code.

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 it will load in any way. The code for the second stage is located 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. The second stage is always on track 0 and block 0 of the named volume, so it can be accessed easily by the first stage.

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 Operating System from a volume that was formatted using the 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 I.8 or iRMX II 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 is in a named file on the bootstrap device. Both the third stage and the application system to be loaded must be in the same directory on the volume.

There are three types of third stages: a generic third stage, a device-specific third stage and a remote third stage. The type needed for your system depends on the size of the application system you intend to load, and if you are booting from a local or remote disk.

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. It then switches the processor into protected virtual address mode and passes control to the application.

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, or the iSBC 221 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 the 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 are located outside the first megabyte.

Remote Third Stage

The remote third stage is similar to the generic third stage in that it only runs in real address mode and can only load applications in the first 840K of memory. This third stage works with the iSBC 552A first stage device driver to load the operating system from a remote device using the Ethernet Local Area Network (LAN).

The remote third stage is different from the generic third stage in that it is loaded by a combination of the first stage iSBC 552A device driver and the firmware executing on the iSBC 552A board, not the second stage.

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 now using. Name the copy so 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 booting. 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. This name was chosen for iRMX II systems to enable it to use the defaults in the Intel microcomputer system firmware.

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.

If your loadfile is located on a remote system, you do not specify the actual name of the load file. Instead, you must specify a word value called the class code. The remote system must create a class code file which contains the mapping of class code values to sets of files to be loaded. In addition, the files to be loaded from a remote system must be processed with utilities provided with the iRMX-NET software product before they can be loaded. Refer to the *iRMX®-NET Software Installation and Configuration Guide* for a description of preparing an application system to be remote booted.

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 and remote third stage include no drivers of their 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, iSBC 215G, and iSBC 221 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.
iSBC 552A Ethernet Controller**	Real Mode Only.
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.

* No SCSI device driver is supplied in the iRMX Bootstrap Loader for the iSBC 386/258 board. You must use the MSA Bootstrap Loader on the iSBC 386/258 board.

** The iSBC 552 board cannot be used in either the remote boot consumer system or the remote boot server system: you must use the newer iSBC 552A board for any facet of remote booting on MULTIBUS I.

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 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 iSDM R3.2 Monitor reserves memory locations 0FE400H to 0FFF7FH for the Bootstrap Loader. The first stage data and stack are located 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.

The remote third stage is located by default at 0BC000H and uses 4K bytes of memory. It also uses an absolute location in memory for an indirect jump instruction. The default absolute memory location is 1050H and is 32 bytes in length.

You can change the location of the remote third stage by using the BR3.CSD SUBMIT file to generate your own version. Chapter 4 describes the BR3.CSD SUBMIT file.

1.5 CONFIGURING YOUR OWN BOOTSTRAP LOADER

If you intend to create your own version of the Bootstrap Loader, you use the Bootstrap Loader configuration and generation files supplied by Intel. These files are located in the directory /BSL in both iRMX I and iRMX II systems. This is a change from previous releases of iRMX 86 where they were located in /RMX86/BOOT and from previous releases of iRMX II where they were in /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.

USING THE BOOTSTRAP LOADER 2

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 main 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 you are using the Intel-supplied first stage, you can enter the load file specifications 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, you can also use the Debug option to pass control to the monitor after loading is complete.

2.2.1 Specifying the Load File

An operator can specify a load file when the first stage of the Bootstrap Loader displays 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. For information on using the D-MON386 Monitor refer to Appendix C.

When entering the monitor's B command or responding to the Bootstrap Loader's asterisk prompt, you 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

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 the filename extension (assumed to be .286). For loading from a remote system, enter the class code value. If you omit this name, the Bootstrap Loader attempts to load the files associated with the default class code from a remote system. The default class code for loading from remote systems is 4000H. A space must precede the class code.

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 enter the boot command.

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

Example: Assume that an iRMX I application system is in the file /BOOT86/MYSYS on drive :WF0:. You can boot this system by entering the following command at the iSDM monitor prompt:

```
.b :wf0:/boot86/mysys
```

For an example using the D-MON386 monitor, refer to Appendix C.

2.2.2 Debug Option

If the iSDM monitor is present in the system, you can include a debug option when specifying a load file. This option instructs the Bootstrap Loader to do the following immediately after loading is complete:

- Set the CPU registers as described in Appendix A so the iRMX Operating System can detect that the debug option was set. iRMX will then execute an interrupt 3 instruction to enter the monitor. 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, displays its prompt and waits for a command from the terminal. (The monitor prompt is a single period for real-mode iRMX I systems or two periods for protected-mode iRMX II systems.) At this point you 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 do 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:

```
.bd
.b d
.bd /boot86/28612
.b  d  :w0:boot86/28612
```

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. These systems display the Bootstrap Loader's first stage asterisk (*) prompt. Place the "D" and at least one space in the command line before the load file specification. Examples of this are:

```
*d
*  d
*d /boot86/28612
* d  :w0:boot86/28612
```

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 display an asterisk prompt so you can enter the name of the load file. To configure the first stage for standalone operation, refer to Chapter 3.

2. Configure the iSDM monitor to include the Bootstrap option and reconfigure the first stage of the Bootstrap Loader to include the first stage device driver(s) needed for bootstrap loading. This is necessary because not all 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 begin bootstrap loading via the monitor's 'B' (boot) command. To use this method, refer to Appendix B 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 will be done 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 usually occurs in response to an external event (such as a system reset or a monitor command), an executing program can also initiate it. 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. (Chapter 3 describes the BS1.A86 and BS1MB2.A86 files.)

   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. To comment out a line, precede it with a semicolon. For more details on absolute code, refer to the *iAPX 86, 88 Family Utilities Guide*.

Refer to Chapter 3 for more information on the BS1.CSD file.

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 types of third stages, depending on the type of system you are loading.

- A generic third stage
- A default device-specific third stage
- A remote 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 elements to consider when you choose a third stage are the size of your system, the type of mass storage devices used 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:

1. the default device-specific third stage regardless of the size of your system, or
2. a default generic or remote third stage for systems up to 840K bytes.

All third stages are supplied for 80286 and 386™ 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 processor's protected mode. For information on writing a third stage driver, refer to Chapter 6.
- If your application will be loaded from a remote device, you must use the remote third stage.

NOTE

The 840K byte limit on systems loaded by the generic and remote third stages 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 Diskettes. This table enables you to pick the appropriate third stage for your system. After you install your system, these files are available in the /BSL directory.

Table 2-1. Supplied Third Stage Files

CPU Board	Device-Specific Third Stage	Generic Third Stage	Remote Third Stage
iSBC 286/10	28612	28612.GEN	28612.REM
iSBC 286/10A	28612	28612.GEN	28612.REM
iSBC 286/12	28612	28612.GEN	28612.REM
iSBC 386/12	38612	38612.GEN	38612.REM
iSBC 286/100A	286100A	286100A.GEN	
iSBC 386/2X	38620	38620.GEN	38620.REM
iSBC 386/3X	38620	38620.GEN	38620.REM
iSBC 386/116*	386100	386100.GEN	
iSBC 386/120*	386100	386100.GEN	

* This usage of any third stage with these CPU boards is possible only if they do not use the MSA Bootstrap Loader.

CONFIGURING THE FIRST STAGE 3

3.1 INTRODUCTION

There are three stages to the Bootstrap Loader. 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:

1. Create a directory in which to generate the first stage and attach it as your default directory.
2. Invoke a SUBMIT file to obtain a local copy of the configuration source files.
3. Edit one or more assembly language source files to include the configurable options and device drivers in the first stage.
4. Edit, then invoke 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 /BSL directory during iRMX installation. Do not modify these files in the /BSL directory. Obtain a local copy by creating a directory in your :home: directory. An example for a 286/12 System is:

```
CREATEDIR :HOME:28612.BSL
ATTACHFILE :HOME:28612.BSL
SUBMIT /BSL/SETUP
```

3.1.1 First Stage Configuration Files

BS1.A86	For MULTIBUS® I systems, 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.
BS1MB2.A86	For a MULTIBUS® II system which does not use the MSA Bootstrap Loader, 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.
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 B552A.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 and link the resulting modules (and any others that you supply). It then locates 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 edit your local copy of either the BS1.A86 or BS1MB2.A86 configuration file (depending upon your system), possibly 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 about the device drivers 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. The individual file descriptions discuss specific areas that you should check.

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    bs1

$include(:f1:bcico.inc)
$include(:f1:bmb2.inc)
$include(:f1:bmps.inc)
$include(:f1:bs1.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 and iSBC 386/12 Channel A
;serial_channel(8274,0d8h,2,8254,0d0h,2,2,8)

;   iSBC 286/10(A)/12 and iSBC 386/12 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)
%device(r0, 0, deviceinit552A, deviceread552A)

%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:bs1.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 386/116/120
;serial_channel(82530,084H,2,82530,084H,2,0,0eh,a)

;   iSBX 354 Channel B (on iSBX #0) for iSBC 386/116/120
;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. 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 finds 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 iSBC 386/116- and iSBC 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 iSBC 386/2X and iSBC 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)

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 executes 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 executing 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	386® 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, 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 an iSBC 386/116- or 386/120 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 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. A value of NONE places no initialization value 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 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 displays an asterisk (*) prompt at the console terminal and waits for you to enter the pathname of the file to load. It tries once to load the file you specify.
  - -- If you enter a pathname, the Bootstrap Loader loads the file with that pathname.

- -- If you enter a CARRIAGE RETURN only, the file with the default pathname loads.

%MANUAL only

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

- The Bootstrap Loader displays an asterisk (*) prompt for a pathname at the console terminal.
- The Bootstrap Loader chooses a device depending on your response.
  - -- If you enter a device name, 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 you enter only a CARRIAGE RETURN, 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 your 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 only

(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 displays an asterisk (*) prompt for a pathname at the console.

- If you respond 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 you respond with a pathname containing a device name, the Bootstrap Loader tries once to load the file you requested.
  - -- If you entered a pathname is, it tries to load the file with that pathname.
  - -- If you entered only a CARRIAGE RETURN, it tries to load the file with the default pathname.

Whenever the Bootstrap Loader's asterisk prompt appears, you can include a Debug Switch along with a device and/or filename specification. Chapter 2 describes the Debug Switch.

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, the Bootstrap Loader first stage assumes a NULL pathname. 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 assign /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 you can use it 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 done by standalone CI and CO routines; that is, routines that are not part of the monitor. If you include the CICO macro, you must do 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 do 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 386/116/120 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). Here, 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 do 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 the 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 do 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, do 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, do 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 determines if the calculated count value is acceptable, 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. Enter this name 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 *iRMX® I* or the *iRMX® II Interactive Configuration Utility Reference Manual* for more information about configuring the Operating Systems.

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 *iRMX® 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 do 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 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
MSC Specific Driver*	deviceinitmsc	devicereadmsc
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
iSBC 552A Driver**	deviceinit552A	deviceread552A

* The MSC drivers support the iSBC 214, iSBC 215G, iSBC 220 and iSBC 221 controllers, and the iSBX 218A controller mounted on the iSBC 215G board. The drivers must be reconfigured to support the iSBC 220 controller.

** The iSBC 552 board cannot be used in either the remote boot consumer system or the remote boot server system: you must use the newer iSBC 552A board for any facet of remote booting on Multibus I.

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 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. 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,

1. 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.

2. 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.

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

;console
;text
%list

;again
;intl
%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 four 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 included automatically.

3.3.3 %LIST Macro

The %LIST macro causes the Bootstrap Loader to display a list of the included device-units at the console whenever you enter 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.

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.

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, and 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, B552A.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, 552A or SCSI, depending on the device driver.

The number and type of parameters that are included with each 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 iRMX Start-up Systems.

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

3.4.1 %B208 Macro

The %B208 macro is used to configure the Bootstrap Loader driver for the iSBC 208 controller board.

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 and %B220 macros are used to configure the Bootstrap Loader driver for the iSBC 214, iSBC 215G, iSBC 220 and iSBC 221 controller boards.

The BMSC.A86 file contains two macros, %BMSC and %B220. However, you can use only one. If you have one of the drivers listed in the Note 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 is used to configure the Bootstrap Loader driver for the iSBX 218A controller board.

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 is used to configure the Bootstrap Loader driver for the iSBC 186/224A controller board.

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. Enter it 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. At present, sector-slipping is unsupported; therefore, set this value 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 is used to configure the Bootstrap Loader driver for the iSBX 251 controller board.

```
%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 is used to configure the Bootstrap Loader driver for the iSBC 254 controller board.

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 is used to configure the Bootstrap Loader driver for the iSBC 264 controller board.

%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 %B552A Macro

The B552A macro is used to configure the Bootstrap Loader driver for the iSBC 552A controller board.

The B552A macro has the form

```
%B552A (boot_addr_list_count, boot_addr_list, boot_port_list_count,
        boot_port_list, indirect_address_cs, def_class_code)
```

where:

boot_addr_list_count	The number of entries in the list for the different possible iSBC 552A memory addresses given in the boot_addr_list. For a particular version of the iSBC 552A firmware only one of these addresses will be valid and only one is required.
boot_addr_list	The list of addresses for each entry delimited by a comma. Each entry is a word. The iSBC 552A controller in Intel System 300 Microcomputers uses address 1040H. (The default value of 104H represents absolute address 1040H.)
boot_port_list_count	The number of entries in the list for the different possible iSBC 552A I/O ports given in the boot_port_list field. For a particular version of the iSBC 552A firmware only one of these addresses will be valid and only one is required.
boot_port_list	The list of addresses for each entry delimited by a comma. Each entry is a word. The iSBC 552A controller in Intel System 300 Microcomputers uses port address 8B4H. Old versions of the Intel Microcomputers used port address 8A4H so both are included in the list. The iSBC 552A bootstrap driver will check each of the combinations of the "port" (boot_port_list) and "address" (boot_addr_list) to determine if the iSBC 552A is present. This allows one set of PROMs to be placed on CPU boards which can be used in systems where the configuration of the iSBC 552A changes.
indirect_address_cs	The base portion of the Real mode address which is used to pass control to the third stage. This is necessary since the first stage does not know where the third stage will be located when the first stage is configured. (The default value 106H represents absolute address 1060H.)

def_class_code	The default remote class code assumed for auto booting. (04000H is the default class code for iRMX II.)

The default invocation of this macro in the B552A.A86 file is

```
%B552A (1, 104H, 2, 8B4H, 8A4H, 106H, 4000H)
```

3.4.9 %BSCSI Macro

This macro allows you to specify the details of a SCSI host board, such as the iSBC 186/03A, 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. Set it to zero.
reserved	Reserved for future use. Set it 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 DMA controller's registers.

dma_separation — A BYTE that indicates the number of bytes separating consecutive ports on the DMA 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 286/100A 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, 2, 0, 0200H, 2, 1, 0D1H)

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

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 286/100A 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 actual device(s) in use.

3.4.10 %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. 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 286/100A 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_x1410a, 0ch, 8, 1, 32h, 6, 0, 0b4h, 0, 0, 0bh)
```

3.4.11 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, take 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 such 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 does the assembly, linkage, and location of the first stage of the Bootstrap Loader. Often you will need to modify it 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 /bsl as :f1:
;
asm86 %2.a86     macro(100) object(%2.obj)    print(%2.lst)
asm86 bserr.a86  macro(50) object(bserr.obj) print(bserr.lst)
;
asm86 b208.a86   macro(50) object(b208.obj)  print(b208.lst)
asm86 bmsc.a86   macro(50) object(bmsc.obj)  print(bmsc.lst)
asm86 b218a.a86  macro(50) object(b218a.obj) print(b218a.lst)
asm86 b251.a86   macro(50) object(b251.obj)  print(b251.lst)
asm86 b254.a86   macro(50) object(b254.obj)  print(b254.lst)
asm86 b264.a86   macro(50) object(b264.obj)  print(b264.lst)
asm86 552A.a86   macro(50) object(552a.obj)  print(552a.lst)
asm86 bscsi.a86  macro(50) object(bscsi.obj) print(bscsi.lst)
;
```

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

```
;   Multibus II configuration
;
;asm86 b224a.a86  macro(50) object(b224a.obj) print(b224a.lst)
;
link86                              &
   %2.obj,                    &
   bserr.obj,                 &
   bcico.obj,                 & ;for standalone serial channel support
   b208.obj,                  &
   bmsc.obj,                  &
   b218a.obj,                 &
   b251.obj,                  &
   b254.obj,                  &
   b264.obj,                  &
   b224a.obj,                 &
   bscsi.obj,                 &
   :f1:bs1.lib                &
to %2.lnk print(%2.mpl) &
&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 %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(%2.mp2)                                    &
   ;bootstrap
;
;      Remove the ';' from the line ';bootstrap' when generating a
;      a standalone Bootstrap Loader in PROM for all configurations
;      except those for 386-based CPU boards.
;*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
;   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

1. Obtain a local copy of the configuration files as described at the beginning of this chapter.
2. Make any changes you might need to the local version of those files.

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, iSBC 186/224A, iSBC 552A, and SCSI device drivers.

```
link86                              &
   %2.obj,                      &
   bserr.obj,                   &
&  :f1:bcico.obj,               & ;for standalone serial channel support
&  b208.obj,                    &
   bmsc.obj,                    &
&  b218a.obj,                   &
&  b251.obj,                    &
&  b254.obj,                    &
   b264.obj,                    &
&  b224a.obj                    &
&   b552a.obj,                   &
&  bscsi.obj,                   &
   :f1:bs1.lib                      &
to %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 all except MSC and 264 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.

3.5.2 Invoking the BS1.CSD Submit File

After you have obtained a local copy of the configuration files and have modified the BS1.CSD file to correspond to your configuration:

1. Invoke the submit file to assemble the Bootstrap Loader files.
2. Link them together.
3. Assign absolute addresses.

The format for invoking the submit file is as follows:

```
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 file BS1.A86 to be assembled and 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 file BS1MB2.A86 to be assembled and 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:

- `SUBMIT BS1( 0FE400H, 0B8000H, bs1 ) <CR>`

- `SUBMIT BS1( 0FE400H, 0B8000H, bs1mb2 ) <CR>`

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. You also have the option of using bigger PROMs.

CONFIGURING THE THIRD STAGE 4

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 three different types of third stages that can be used to load iRMX II files: the generic third stage, the device-specific third stage and the remote third stage. All load OMF-286 modules, but the generic and remote third stages leave the processor in real address mode while they load. 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, remote or device-specific third stage. However, the differences are small enough that all 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, remote, or device-specific third stage) is very similar to configuring the first stage. It involves the following operations:

1. Create a directory in which to generate the third stage and attach it as your default directory.
2. Invoke a SUBMIT file to obtain a local copy of the configuration source files. If you did this step during configuration of the first stage, you do not need to repeat it.
3. Edit 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.
4. Invoke 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, as you do for the first stage. The SUBMIT file assembles and links the device configuration files, and you do not 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.

The remote third stage does not use any device drivers, real or protected mode. When the first stage transfers control to the remote third stage, the entire application system has been loaded into the host's memory. The configuration information supplied about the remote third stage does not specify device parameters, it only specifies data for the third stage that it could not otherwise obtain.

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

BS3.A86 BS3MB2.A86 BG3.A86 BR3.A86	These assembly language source files contain macros that specify the devices supported by the third stage (for device-specific third stage only). They identify the CPU board and tell 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. The BG3.A86 file applies to the generic third stage on either MULTIBUS I or MULTIBUS II systems. The BR3.A86 applies to the remote third stage on MULTIBUS I systems.
BMSC.A86 B264.A86	These assembly language source files apply only 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 BR3.CSD	These SUBMIT files contain the commands you need 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. The BG3.CSD file applies to the generic third stage and the BR3.CSD file applies to the remote 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, remote, and generic third stages.

4.3 BS3.A86, BS3MB2.A86, BG3.A86, AND BR3.A86 CONFIGURATION FILES

Figures 4-1, 4-2, 4-3 and 4-4 list the assembly language configuration files for the device-specific third stage files BS3.A86 and BS3MB2.A86, the generic third stage file BG3.A86, and the remote third stage file BR3.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, BG3.A86 or BR3.A86 file to include or exclude macros. For each macro, a percent sign (%) preceding the name includes (invokes) the macro. 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, BG3.A86 and BR3.A86 files. Unless otherwise specified, the macros can appear in either of the three files (the %DEVICE macro is the only one that applies only 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,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 (:fl:bs3cnf.inc)
;
;intl
%int3
;halt

%cpu board (286/12)

%installation(n)

%end
```

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

```
$include (:fl:bs3cnf.inc)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                         Remote devices
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%remote_device (r0,thirdstage552A, data_552A)
;
%int3
;halt
;
%cpu_board (286/12)

%end
```

Figure 4-4. Intel-Supplied BR3.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 following:

- The base address of the Message Passing Coprocessor (MPC)
- The address distance between MPC ports
- 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
iSBC 552A	third stage 552A	N/A	data_552A

4.3.3 %REMOTE_DEVICE

The %REMOTE_DEVICE macro applies only to the remote third stage (BR3.A86 file). It associates a device with a particular third stage device driver. The syntax of the %REMOTE_DEVICE macro is as follows:

```
%REMOTE_DEVICE (name, device$driver, device$data)
```

where:

name: The name of the device you use to invoke the Bootstrap Loader. You should always specify the same name that you used for the device in the BS1.A86 file.

device$driver: Public name of the remote third stage device driver. For the iSBC 552A, this name is thirdstage552A and cannot be changed.

device$data: Name of the driver's data segment. For the iSBC 552A, this name is data_552A and cannot be changed.

4.3.4 %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. SCSI controllers do not require this information. 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 SASI controller board.

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 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)
```

4.3.5 %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 the third stage configuration files are assembled.

4.3.6 %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 the third stage configuration files are assembled.

4.3.7 %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 the third stage configuration files are assembled.

4.3.8 %CPU_BOARD Macro

The %CPU_BOARD macro specifies the type of processor board in your system. The third stage needs this information so 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:

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

4.3.9 %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 loads and then executes with no monitor entry in between. If it is 'y', the monitor is entered after the system loads. 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 instructing you how to proceed with the loading:

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

This is an example. The actual message will differ according to the type of system you are booting.

NOTE

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

4.3.10 %END Macro

The %END macro is required at the end of the BS3.A86, BS3MB2.A86, BG3.A86, and BR3.A86 files. The syntax of this macro is as follows:

```
%END
```

There are no parameters associated with the %END macro.

4.3.11 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, take 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 such 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

SUBMIT files (BS3.CSD and BG3.CSD) are used to generate the two types of third stages. BS3.CSD does the assembly, linkage, and location of the device-specific third stage. BG3.CSD does the same operations for the generic third stage. Figures 4-5, 4-6 and 4-7 show the Intel-supplied BS3.CSD and BG3.CSD files.

```
attachfile bsl as :f1:
;
asm86 %0.a86
asm86 bmsc.a86
asm86 b218a.a86
asm86 b264.a86
asm86 bscsi.a86
asm86 b224a.a86
;
link86              &
   %0.obj,      &
   bs3.lib,     &
   bmsc.obj,    &
   b218a.obj,   &
   b264.obj     &
&  bscsi.obj,   &
&  b224a.obj    &
to %0.lnk print(%0.mp1) notype nolines nosymbols
;
loc86 %0.lnk                        &
    addresses(classes(code(%1)))      &
    order(classes(code,data))         &
    noinitcode purge                  &
    start(bs3)                        &
    map print(%0.mp2)
;
;   The Third Stage, located at address %1, is in the file %0
;
```

Figure 4-5. Device-Specific Third Stage SUBMIT File (BS3.CSD)

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

Figure 4-6. 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 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 bsl AS :F1:
```

to

```
;ATTACHFILE bsl AS :F1:
```

```
attachfile /bsl as :f1:
;
asm86   %0.a86
;
link86                  &
    %0.obj,         &
    :f1:br3.lib     &
to  %0.lnk print(%0.mpl) notype nolines nosymbols initcode
;
loc86   %0.lnk to %0.loc                                &
    addresses(segments(code(%1)))   noinitcode  &
    order(segments(CODE,DATA,DATA_ENTP))            &
    segsize(stack(200H)) start(106H,0)          &
    objectcontrols (purge)                          &
;
;   Convert the located module to remote boot format.
;   The Xlate86 program is supplied with iRMX-NET,
;    Release 3.0 or newer
;
delete %2
xlate86 %0.loc over %2 r a n d
;
;
```

Figure 4-7. RemoteThird Stage SUBMIT File (BR3.CSD)

4.4.2 Invoking the Submit File

After you have modified either the BS3.CSD, BG3.CSD or BR3.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.

Remote third stage

SUBMIT BR3 (filename, third_stage_addr, output_filename)

where:

filename	The name of the third-stage configuration file you are using (BS3.A86 for MULTIBUS I systems).
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.
output_filename	The final name of the remote third stage.

This submit file invokes the Xlate86 utility which is provided with iRMX-NET or INA 960. This requires that you have already installed the Xlate86 utility.

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 invoke to change the memory assignments.

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

Description	Default	Maximum Size*	Configuration File
1st Stage Code	Application Dependent * 0FE400H for iSDM R3.2	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
3rd Stage (remote) Code	0BC000H	4K Bytes	BR3.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 Diskettes contain 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.

WRITING A CUSTOM FIRST-STAGE DRIVER 5

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 plan to write a driver for the MULTIBUS® II System Architecture (MSA) Bootstrap Loader, refer to the *MULTIBUS® II System Architecture (MSA) Bootstrap Loader Specification* rather than this manual.

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 /BSL/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 a MULTIBUS II environment which does not use the MULTIBUS II System Architecture bootstrap protocol, 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. Later 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$init: PROCEDURE (unit) WORD PUBLIC;
DECLARE unit        WORD;
    .
    .   (code as described below)
    .
END device$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 do 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

Besides 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 non-MSA 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$ptr             POINTER;
        .
        .   (code as described below)
        .
END device$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 do 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

Besides 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. BSERROR takes the action specified in the file BSERR.A86. None of these actions return to the caller.

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 that the 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_newdrvl  dw  100H    ;MSC wakeup address

device_newdrvl  db  0

drtab_newdrvl   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. 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_newdrvl  word   ;Wakeup port
                                  ;address
   extrn   device_newdrvl  byte   ;Device number
   extrn   drtab_newdrvl   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 NEWDRV1.INC to define a macro for the device-specific parameters declared in Step 1.

```
%*DEFINE (bnewdrv1(wakeup,ncyl,nfsur,nrsur,nsec,secsize,nalt)) (
         name bnewdrv1
         public   wakeup_newdrv1, device_newdrv1, drtab_newdrv1
code_newdrv1   segment byte public 'CODE'
wakeup_newdrv1 dw      %wakeup

device_newdrv1 db      0

drtab_newdrv1  dw      %ncyl
               db      %nfsur
               db      %nrsur
               db      %nsec
               dw      %secsize
               db      %nalt

code_newdrv1   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 NWDRV1.A86 could contain the following information to invoke the macro defined in Step 2.

```
 $include(:f1:newdrv1.inc)

 %bnewdrv1(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 /BSL where the Bootstrap Loader configuration files are located. For example, the following Human Interface commands can be used to copy the files created in Steps 2 and 3.

```
- copy newdrv1.inc, nwdrv1.a86 to /bsl

  newdrv1.inc copied to /bsl/newdrv1.inc
  nwdrv1.a86 copied to /bsl/nwdrv1.a86
```

5. Create a directory in which to generate the first stage and attach to it. Then use the SUBMIT file /BSL/SETUP.CSD to obtain a local copy of the configuration files. Also copy your new configuration source file, NWDRV1.A86. Because the BS1.CSD SUBMIT file attaches /BSL as :F1:, NWDRV1.A86 will reference /BSL/NWDRV.INC.
6. 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. 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 bs1.a86    macro(90) object(bs1.obj)   print(bs1.lst)
          .
          .
          .
    asm86 b264.a86   macro(50) object(b264.obj)  print(b264.lst)
    asm86 bscsi.a86  macro(50) object(bscsi.obj) print(bscsi.lst)
--> asm86 nwdrv1.a86 macro(50) object(nwdrv1.obj print(nwdrv1.lst)
   ;
   link86                     &
      bs1.obj,                &
      bserr.obj,              &
   &   :f1:bcico.obj,         &;standalone serial channel support
          .
          .
          .
      bscsi.obj,              &
      b264.obj,               &
-->   nwdrv1.obj,             &
      :f1:bs1.lib             &
   to bs1.lnk print(bs1.mp1) &
          .
          .
          .
```

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 *iRMX® II Nucleus User's Guide*.

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. As a result, 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,
- 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 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.

5.5.2 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 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 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 1F4H 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;
    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.

5.5.3 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 call this utility before attempting any other Bootstrap Loader Communication System utility calls.

Condition Codes

This utility has no condition codes.

5.5.4 Send and Receive Transaction Models

Besides 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 send a message with no request for a response., You can also call them 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 1F4H 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;
                    .
    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 for, 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 1F4H 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 simultaneously 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$id of 1F4H.  During message             *
 *     broadcasting, the host$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 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 non-rsvp 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 between 0000H and 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 value
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 betweem 0000H and 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 do 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 particular 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(80386)

      ;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.

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 be in the file NEWDRV1.A86.

```
- asm86 newdrv1.a86 object(newdrv1.obj)
iRMX <I/II> 8086/87/186 MACRO ASSEMBLER, V < >
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 is in the directory /BSL. The following commands add the object modules generated in Step 1.

```
- LIB86
iRMX <I/II> 8086 LIBRARIAN V < >
Copyright <    > INTEL CORPORATION
*add newdrv1.obj to /bsl.lib
*exit
```

3. Attach the directory containing your local copy of the Bootstrap Loader configuration files as the current default directory:

```
- attachfile :home:mybsl
:home:mybsl 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 0C000H as the memory location of the first stage and 0BC000H as the memory location of the second stage.

   ```
   - submit bs1(0C000H, 0BC000H)
   ```

   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 until BS1.LIB is upgraded in a future release of the iRMX Bootstrap Loader.

WRITING A CUSTOM THIRD-STAGE DRIVER 6

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, there is no need for you to create new device drivers 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 help you in writing your own drivers, your iRMX Operating System package contains the source code for a working third stage driver. After installing your iRMX system, you can find the source code in the file /BSL/BPMSCG.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 be 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 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 if 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 a non-MSA 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 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$init: PROCEDURE (unit) WORD PUBLIC;
DECLARE unit WORD;
    .
    .   (code as described below)
    .
END device$init;
```

where:

device$init	The name of the device initialization procedure. You can choose any name you wish for this procedure, as long as the names of other third-stage procedures do not conflict.
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 do 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 obtain the device granularity. (For information on the location and organization of the volume label, see the *iRMX® 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

Besides 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 the same. 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. Here, 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. Then, 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$ptr POINTER;
    .
    .   (code as described below)
    .
END device$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 do similar functions. Therefore, you may want to create the third-stage read procedure by doing modifications on the first-stage read procedure (if, for instance, it has already been written and is located in PROM). If the first-stage read procedure does not yet exist, you can write the third-stage read procedure first. 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 do 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 do 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 do 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 do 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 with this task, Intel provides a small, single-thread communication system. This 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.

For a more complete description of Bootstrap Loader Communication System concepts similar to Nucleus Communication System concepts, refer to the *iRMX® II Nucleus User's Guide*.

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, do 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 the device read and device initialization procedures in the file NWDRV3.A86.

```
- asm86 nwdrv3.a86 object(nwdrv3.obj)
iRMX II 8086/87/186 MACRO ASSEMBLER, V <  >
Copyright <years> 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 is in the directory /BSL. The following commands add the object modules generated in Step 1.

```
- LIB86
iRMX II 8086 LIBRARIAN V <  >
Copyright <years> INTEL CORPORATION
*add nwdrv3.obj to /bsl/bs3.lib
*exit
```

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

```
- attachfile :home:mybsl
:home:mybsl, 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.

```
- submit BS3(new3stg, 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.

ERROR HANDLING 7

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 if 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 third stage configuration 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). This is because the generic third stage supports all the devices supported by the first stage. If you invoke the Bootstrap Loader with a device name 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, 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 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 board

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 multiple 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 a protected mode 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 7-1 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 on entering the monitor, 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 type 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 type 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® Human Interface* for more information about the condition codes.

AUTOMATIC BOOT DEVICE RECOGNITION A

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 combines information found in other iRMX 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 (the debug switch was set).
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 "ABDR" screen).
6. The Extended I/O System attaches the device as the system device. It assigns it the logical name that you must have specified during the configuration of the Extended I/O System (DLN prompt on the "ABDR" 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 or
                                                      :SD:RMX86/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. The Extended I/O System creates the logical name you specify only if you set ABR to "yes." Intel recommends not changing from the default.

  With the DPN prompt, you specify the physical name of a device you want to use as your system device if the Extended I/O System cannot find the name RQBOOTED catalogued in the root object directory. This 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. Then, the Extended I/O System uses the device name specified in the DPN prompt as the system device.

With the DFD and DO prompts, you set other characteristics associated with the system device. For most cases, the defaults (DFD=Named and DO=0000H) are the preferred values.

```
(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. However, 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 *iRMX® Interactive Configuration Utility Guide* 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 **boldfaced** 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

PROMMING THE BOOTSTRAP LOADER AND THE iSDM™ MONITOR B

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. Attach the directory in which you generated the first stage as your default directory.
2. Enter the name of the (version 1.4 or later) software used with the iUPP Universal PROM Programmer:

   ```
   ipps
   ```

3. Specify that the PROMs are 27128 EPROM devices:

   ```
   type 27128
   ```

4. Initialize the file type to be loaded:

   ```
   initialize 86
   ```

   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.)

```
format :f1:bs1(0FE400H,0FFF7FH)
3
2
1
0 to bs1.evn
1 to bs1.odd
<cr>
```

6. Program the PROMs with the iSDM monitor before you program the Bootstrap Loader. This process is described in the *iSDM™ System Debug Configuration and Installation Manual.*

7. 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:

```
copy :f1:bs1.evn to prom(3200H)
```

8. Do the same for the odd-numbered bytes.

```
copy :f1:bs1.odd to prom(3200H)
```

9. Exit the IPPS program.

```
exit
```

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 = 0F200H

D-MON386 MONITOR C

C.1 OVERVIEW

Intel Modules Development Platforms are delivered with the Bootstrap Loader's first stage, the D-MON386 monitor, and the SCT in the PROM devices.

You can specify a load file:

- When the monitor has issued a prompt. In this case, you can enter the monitor's BOOT (bootstrap) command, followed by the name of the load file (include the name within double quotes if you are using the D-MON386 monitor).

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

':device:pathname' (D-MON386)

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: 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:

```
boot ':wf0:/system/mysys'
```

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of a Intel product users. This form lets you participate directly in the publication process. Your commen will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of th publication. If you have any comments on the product that this publication describes, please conta your Intel representative.

1. Please describe any errors you found in this publication (include page number).

2. Does this publication cover the information you expected or required? Please make suggestion for improvement.

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

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

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____ PHONE ( ) _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

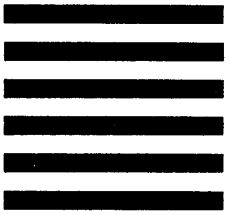
E'D LIKE YOUR COMMENTS . . .

is document is one of a series describing Intel products. Your comments on the back of this form will lp us produce better manuals. Each reply will be carefully reviewed by the responsible person. All mments and suggestions become the property of Intel Corporation.

'ou are in the United States, use the preprinted address provided on this form to return your mments. No postage is required. If you are not in the United States, return your comments to the Intel les office in your country. For your convenience, international sales office addresses are printed on ; last page of this document.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 79 HILLSBORO, OR



POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
OMSO Technical Publications, MS: HF3-72
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-9978

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
Flower-Hill Shin-machi
1-23-9, Shinmachi
Setagaya-ku, Tokyo 15

NETHERLANDS
Intel Semiconductor (Netherland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

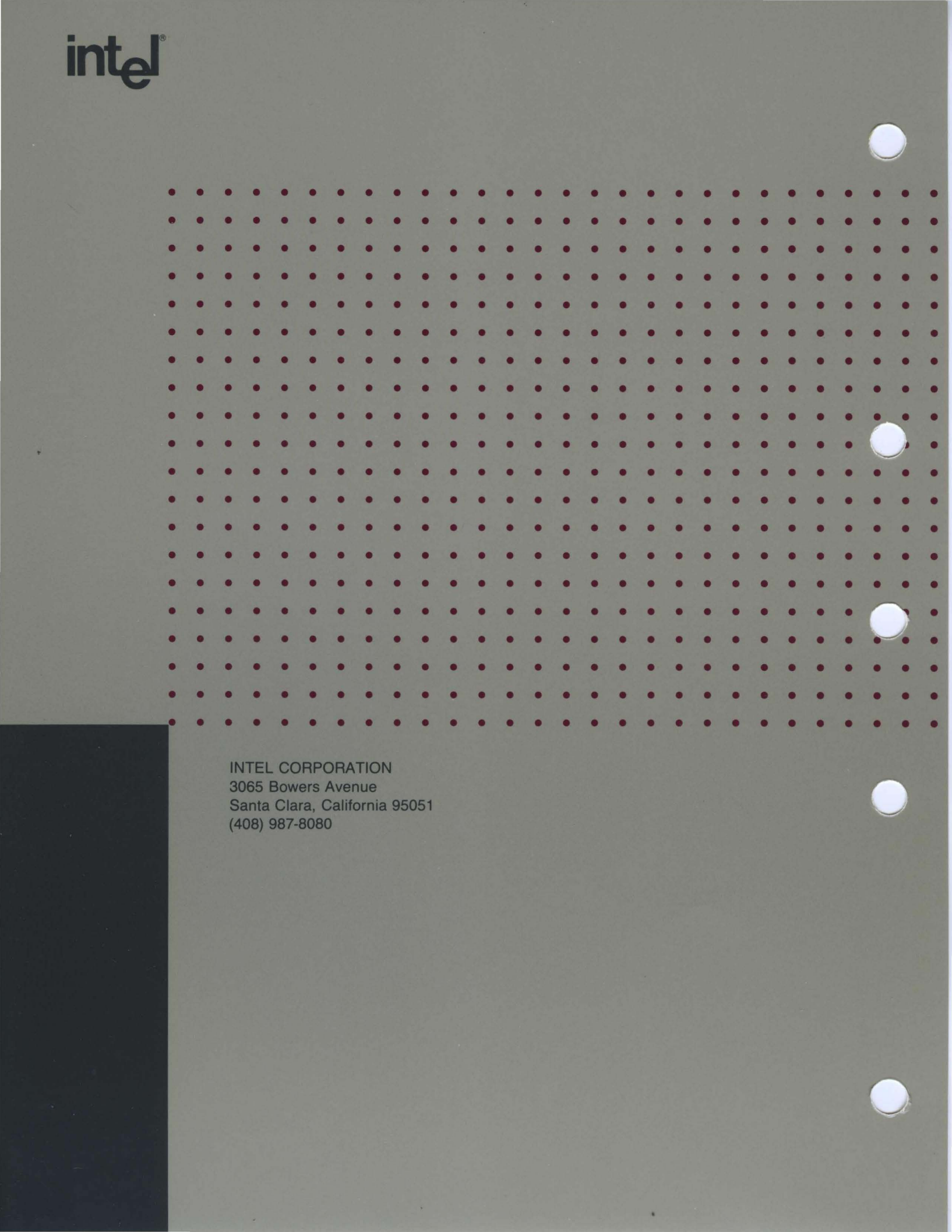
NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor G.N.B.H.
Seidlestrasse 27
D-8000 Munchen



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