iRMX 86™
LOADER REFERENCE MANUAL

Manual Number: 143318-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

REV.	REVISION HISTORY	PRINT DATE
-001	Original Issue — Reflects Release 3.0 of the iRMX 86 Operating System. This manual contains some information that formerly was in the iRMX 86 I/O System and Loader Reference Manual.	5/81

A548/282/3.5K DD

PREFACE

This manual documents the iRMX 86 Bootstrap and Application Loaders. It contains some introductory and overview material, as well as detailed descriptions of the system calls of the Application Loader. The system calls described in this manual can be used by application programmers. Other system calls, reserved for system programmers, are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

READER LEVEL

This manual is written for application programmers who are already familiar with:

- The concepts and terminology introduced in the iRMX 86 NUCLEUS REFERENCE MANUAL.
- The PL/M-86 programming language.
- The concepts and terminology introduced in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.
- The concepts and terminology introduced in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.
- The LINK86 and LOC86 commands and their controls, as described in the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS and in the 8086 FAMILY UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS.

CONVENTIONS

This manual uses a generic shorthand to refer to system calls. For example, A$LOAD is used to refer to RQALOAD. The actual PL/M-86 external procedure names are shown only in Chapter 7, which describes the system calls of the Application Loader.

Although Chapter 7 lists only the PL/M-86 calling sequences, you can invoke the system calls from programs written in assembly language. If you need to use assembly language invocation, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.

RELATED PUBLICATIONS

In several places, this manual refers to other Intel documentation. Wherever such references occur, this manual lists only the title of the document to which reference is being made. The following list provides the document numbers.

Manual	Number
Introduction to the iRMX 86™ Operating System	9803124
iRMX 86™ Nucleus Reference Manual	9803122
iRMX 86™ Basic I/O System Reference Manual	9803123
iRMX 86™ Extended I/O System Reference Manual	143308
iRMX 86™ System Programmer's Reference Manual	142721
iRMX 86™ Configuration Guide	9803126
iRMX 86™ Installation Guide	9803125
Guide to Writing Device Drivers for the iRMX 86™ I/O System	142926
iRMX 86™ Programming Techniques	142982
iRMX 86™ Human Interface Reference Manual	9803202
iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems	121616
8086 Family Utilities User's Guide for 8080/8085-Based Development Systems	9800639
PL/M-86 User's Guide for 8086-Based Development Systems	121636
PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems	9800478

CONTENTS

CONTENTS (continued)

PAGE

CONTENTS (continued)

FIGURE

TABLE

CHAPTER 1. ORAGNIZATION OF THIS MANUAL

This manual is divided into seven chapters. Some of the chapters contain introductory and instructional information, while others contain reference information. The following list will help you decide which chapters to read.

Chapter 1 This chapter describes the organization of the manual. You should read this chapter if you are using the manual for the first time.

Chapter 2 This chapter describes the differences between the Bootstrap Loader and the Application Loader. It also gives examples of applications that could advantageously use the loaders.

Chapter 3 This chapter discusses the features of the Bootstrap Loader. Because the Bootstrap Loader has no system calls, this is the chapter to which you should refer when you have questions about the capabilities of the Bootstrap Loader.

Chapter 4 This chapter provides guidance for writing a device driver to be used with the Bootstap Loader. You should refer to this chapter only if your system must bootstrap load from a device other than those for which Intel has supplied drivers.

Chapter 5 This chapter describes the features of the Application Loader. You should read this chapter before you refer to Chapter 7 for the first time.

Chapter 6 This chapter explains how you can advantageously use the asynchronous system calls of the Application Loader.

Chapter 7 This chapter describes the system calls of the Application Loader.

CHAPTER 2. INTRODUCTION

The iRMX 86 Operating System provides two loaders: the Bootstrap Loader, and the Application Loader. This chapter briefly describes the differences between the two.

INTRODUCTION TO THE BOOTSTRAP LOADER

The purpose of the Bootstrap Loader is to provide a means of loading part or all of your application system from secondary storage into RAM (random-access memory) whenever the iAPX 86 processor is reset. Although this purpose seems remarkably simple, the Bootstrap Loader can provide your application system with a significant amount of flexibility. Let's look at two examples, both of which show how the Bootstrap Loader can reduce your expenses.

BOOTSTRAP LOADER AND SIMPLIFIED MAINTENANCE

After you have developed and manufactured your product (your application system), you distribute the product to "the field." If you are an OEM (original equipment manufacturer) you sell the product to customers, and if you are a VEU (volume end user) you provide the product to your employees or subsidiaries.

In either case, you must be concerned with maintenance. Suppose that after the product has been in use for several years, you find a means of improving it. If your product does not include the Bootstrap Loader, your application software resides in ROM (read-only memory). This means that to make changes to the systems in the field, you must produce new ROM chips that contain the changed software, and you must install the chips in the systems. This is a relatively expensive process because it involves sending engineers to your customers to upgrade the product.

In contrast, if your product does incorporate the Bootstrap Loader, you need not manufacture and install new ROM chips. Instead, you can place the revised software on flexible diskettes and mail the diskettes to your customers (if you are an OEM) or employees (if you are a VEU). They simply replace the old diskettes with the new diskettes. Then, whenever they start up the system, the Bootstrap Loader reads the updated software into RAM (random-access memory).

This example shows how the Bootstrap Loader can simplify the process of updating the application system by:

- reducing the number of customer visits you must make.
- eliminating the need to manufacture new ROM chips.

You can use the same technique to distribute corrected software to your customers whenever you correct a bug in your application software.

BOOTSTRAP LOADER AND OPTIONS IN YOUR PRODUCT

Suppose that the hardware of your product can be used for several purposes or applications. For instance, suppose your product consists of one or more flexible diskette drives, a printer, a terminal, and a box containing the iAPX 86 microprocessor, related memory boards, and interface boards. This collection of hardware can be used to construct a word processor, a data base system, a payroll system, a reprogrammable computer, or other application systems. The only difference between all these applications is the software included in the system.

When your customers have the Bootstrap Loader, you can place the application software on a flexible diskette rather than in the system's ROM. Then the only difference between all these systems is the kind of diskette that you sell the customer. If your customers need a word processor, sell them the word processing diskette. If they need a data base system, sell them the data base diskette. If they need both, sell them both diskettes.

INTRODUCTION TO THE APPLICATION LOADER

The purpose of the Application Loader is similar to that of the Bootstrap Loader in that both load code from secondary storage into RAM. The difference is that the Application Loader allows your tasks to control the loading operation.

By allowing your tasks to load programs from secondary storage, the iRMX 86 Application Loader reduces the amount of memory required. Programs that are used only intermittently can remain on secondary storage until they are required. Then one of your tasks can load them and start them running. After a loaded program has finished running, the memory it occupied can be used for other purposes.

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

SUMMARY

The iRMX 86 Operating System provides two kinds of loaders -- a Bootstrap Loader, and an Application Loader:

- The Bootstrap Loader is generally invoked only when the application system starts running. Consequently the Bootstrap Loader does not provide any system calls.

- The Application Loader, which does provide system calls, allows your tasks to load programs from secondary storage into memory. This loader allows large programs to run in systems that haven't enough memory to accommodate the entire program at one time, and it allows programs that are seldom used to reside on secondary storage rather than in primary memory.

If you are interested in obtaining more information about the Bootstrap Loader, refer to Chapter 3. For additional information about the Application Loader, refer to Chapter 5.

CHAPTER 3. USING THE BOOTSTRAP LOADER

The iRMX 86 Bootstrap Loader exists for one purpose. It allows you to store your application software and the large majority of the iRMX 86 software on secondary storage rather than in ROM (read-only memory). Whenever the iAPX 86 microprocessor is reset, the Bootstrap Loader loads the Operating System and the application software into RAM (random-access memory) and transfers control to the Operating System.

In spite of its straightforward reason for existence, the Bootstrap Loader does have a number of features, some of which are optional. You, the OEM or VEU, can decide which of these features are useful for your product. Then, during process of configuring your application system, you can include the features you want and exclude the features you don't want.

TERMINOLOGY OF THE BOOTSTRAP LOADER

The following terms are used frequently in this chapter:

- first and second stage
- device drivers
- file to be loaded
- end user

You must become familiar with these terms in order to understand the rest of this chapter. The following few paragraphs define the terms.

FIRST AND SECOND STAGES

The Bootstrap Loader consists of two parts -- the first stage and the second stage. Only the first stage resides in ROM. It starts running whenever the iAPX 86 microprocessor is reset. The purpose of the first stage is threefold:

- First, it ascertains which secondary storage device contains the file to be loaded.
- Second, it ascertains which file is to be loaded.
- Third, it loads and passes control to part of the second stage.

The second stage resides on secondary storage. Specifically, it resides on the device from which the Bootstrap Loader loads your software. The purpose of the second stage is to complete the bootstrapping process by performing the following steps:

- First, it finishes reading itself into main memory.
- Second, it finds the file to be loaded (the file containing the Operating System and application software).
- Third, it loads the file into main memory.
- Fourth, it transfers control to the loaded file.

The details of the first and second stages are discussed later in this chapter.

DEVICE DRIVERS

The Bootstrap Loader can be used with any kind of secondary storage device. Disks, flexible diskettes, bubble memories, magnetic tapes -- the Bootstrap Loader will work with any of them. However, for each device with which you wish to use the Bootstrap Loader, you must have a device driver.

A device driver is a collection of procedures that allows the Bootstrap Loader to communicate with the device that contains the file to be loaded. Device drivers for the Bootstrap Loader differ from the drivers required by the Basic I/O and Extended I/O Systems. Because of this difference, Chapter 4 of this manual contains instructions for writing device drivers for the Bootstrap Loader.

However, there is an excellent chance that you will not need to write a device driver. The iRMX 86 product includes device drivers for all of the following random-access devices:

- iSBC 204 Flexible Diskette Controller
- iSBC 206 Flexible Diskette Controller
- iSBC 215 Winchester Disk Controller
- iSBX 218 Multimodule Flexible Diskette Controller (single density only) when used with the iSBC 215 controller
- iSBC 254 Bubble Memory Controller

Since these drivers are part of the iRMX 86 product, you need only attach the driver or drivers that you want to the Bootstrap Loader. The method for doing this is discussed in the iRMX 86 CONFIGURATION GUIDE.

FILE TO BE LOADED

The iRMX 86 Bootstrap Loader loads one file from a secondary storage device. This file (which is called the "file to be loaded") must be a named file. (For a description of named files, refer to either the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL or the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.) For information about the creation of this file, refer to the section of this chapter entitled "Requirements of the File to Be Loaded."

END USER

An end user is a person who will be using the application system that you are creating. For instance, if you are building equipment for use in hospitals, your end users are the doctors, nurses, or technicians who will be running the equipment. Throughout this chapter, you will find explanations that correlate your actions during configuration to features provided to your end users.

OPTIONS OF THE FIRST STAGE

The first stage of the Bootstrap Loader consists of two parts. One part is device driver software and the other is the software that loads the second stage. Both parts must reside in ROM.

The amount of memory needed by the device drivers depends upon how many device drivers you choose to include in the Bootstrap Loader. Each driver requires between 300 and 500 (decimal) bytes of ROM with the precise number depending upon the device. The process of writing a device driver for the Bootstrap Loader is discussed in Chapter 4 of this manual. The process of incorporating a device driver into the Bootstrap Loader is discussed in the iRMX 86 CONFIGURATION GUIDE.

The heart of the first stage, the part that loads the second stage, requires between 100 and 500 bytes of ROM, with the exact amount being a function of the options that you choose to include in the Bootstrap Loader. The first stage provides you with four options:

- the location (in ROM) of the first stage
- the location (in RAM) at which the first stage is to load the second stage
- the method to be used for selecting the device containing the file to be loaded
- the method to be used for selecting the file to be loaded

The following sections describe your options in each of these areas. You must specify your choices during the process of configuring the system. For detailed information as to how to configure the first stage of the Bootstrap Loader, refer to the Bootstrap Loader chapter of the iRMX 86 CONFIGURATION GUIDE.

LOCATION OF THE FIRST STAGE

You must decide where in memory you wish to place the first stage. The only restriction is that the first stage must be in ROM. And, if you wish to have the first stage run whenever the iAPX 86 microprocessor is reset, you must use the BOOTSTRAP switch in the LOC86 command when you locate the first stage.

For more details regarding the LOC86 command, refer one of the following manuals:

- If you are using a development system that incorporates an iAPX 86 microprocessor (for example, a Series III development system), refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS.

- If you are using a development system that incorporates only an 8080 or an 8085 microprocessor (for example, a Series II development system), refer to the 8086 FAMILY UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS.

LOCATION OF THE SECOND STAGE IN RAM

The first stage of the Bootstrap Loader loads the second stage from secondary storage into RAM. When you configure the first stage, you must specify where in RAM you want it to load the second stage. There are two points you must consider when you select the location. First, the second stage can be loaded only into RAM that can be accessed by the controller of the bootstrap device. Second, you must avoid a conflict between the memory of the second stage and the memory required to contain the information from the file to be loaded. The reason for the second point is that the second stage is required until the loading process is completed. If the file being loaded overlays the second stage during the loading process, the loading process will not complete successfully.

Be aware that the second stage is no longer needed once the bootstrap loading process has been completed. This means that the memory occupied by the second stage (6144 bytes decimal) can become part of the memory pool for your application system.

In summary, when you specify the location of the second stage during the configuration process, heed the following two rules:

- Place the second stage in RAM locations that are not to be occupied by any information in the file to be loaded. If you fail to heed this rule, the bootstrap loading process will not be successful.

- During the process of configuration, do not reserve the memory occupied by the second stage of the bootstrap loader. By heeding this rule, you will ensure that the memory occupied by the second stage becomes part of the memory pool of your application system.

METHOD TO BE USED FOR SELECTING THE DEVICE

One of the functions performed by the first stage of the Bootstrap Loader is the selection of the device from which the information is to be loaded. The first stage can use any of three methods for selecting the device. During the process of configuring your application system, you must tell the Bootstrap Loader which of the three options to use. The three options are:

- no selection

- automatic selection

- manual selection

The following sections discuss each of these methods.

No Selection

This option means that during the configuration process, you must specify the name of the device from which the file is to be loaded.

From the point of view of your end user, this option means that the bootstrap loading operation always uses a particular device. Whenever your end user attempts to bootstrap load, the Bootstrap Loader will check to see if the device is ready. If it is ready, the loading operation begins. If the device is not ready, the loading operation terminates.

Automatic Selection

This option means that, during the configuration process, you must specify a list of devices that can be used for bootstrap loading. Then, when the Bootstrap Loader is running, it will cycle through the list repeatedly until one of the devices becomes ready. The first ready device that the Bootstrap Loader finds is the device to be used in the bootstrap loading operation.

From the point of view of your end user, this option means that bootstrap loading can involve any of a collection of devices. To select the device, your end user ensures that only one device is ready. Then when the user invokes the Bootstrap Loader, it will load from the sole ready device.

Be aware that if you configure your Bootstrap Loader for automatic selection and you provide a list of only one device, the behavior of the Bootstrap Loader will not be the same as under the no-selection option. The difference is that with the no-selection option the Bootstrap Loader tests the device once. If the device is not ready, the Bootstrap Loader halts. In contrast, with the automatic-selection option, the Bootstrap Loader will test the device repeatedly until the device becomes ready.

Manual Selection

If you select this option, you must still enter a list of devices during the configuration process. The bootstrap loader will use a terminal to find out which device your end user wants to load from. If the end user specifies no device, or if the end user specifies a device not included in your list, the Bootstrap Loader will switch to automatic selection.

From the point of view of your end users, this option means that the Bootstrap Loader will prompt for a device name. The Bootstrap Loader will indicate that it is ready to accept a device name by displaying an asterisk (*) on the terminal.

Once your end users see the asterisk, they can enter the device name surrounded by colons. For example, to select device F0, your end users must enter :F0:.

After your end user enters the device name, the Bootstrap Loader compares the name to the entries in your list of devices. This comparison does not differentiate between upper case letters and lower case letters. For example, if your list includes the device MT0, and your end user enters :mt0:, the Bootstrap Loader would find the device in your list.

In order to use the manual-selection option you (or your end user) must incorporate a terminal in the system. The terminal requires software. To ascertain your options regarding this software, refer to the "How the First Stage Communicates With a Terminal" section of this chapter.

METHOD TO BE USED FOR SELECTING THE FILE TO BE LOADED

One of the functions performed by the first stage of the Bootstrap Loader is the selection of the file from which the information is to be loaded. The first stage can use either of two methods for selecting the file.

During the process of configuring your application system, you must tell the Bootstrap Loader which of the two options to use. The two options are:

- loading a default file
- allowing the end user to specify the file

We will examine each of these options shortly. But before we do, let's look at the requirements of the file to be loaded.

Requirements of the File to Be Loaded

The iRMX 86 Bootstrap Loader loads information from one named data file. If you are unfamiliar with named data files, refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL or to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

The information in the file must be object code in absolute form. However, it can consist of more than one module. For example, it can consist of your application software and the software of the iRMX 86 Operating System.

To combine several absolute modules into a single file you must use the LIB86 command. For information regarding this command, refer to one of the following manuals:

- If your development system is based on the iAPX 86 microprocessor (as is the Series III development system), refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS.
- If your development system is based on an 8080 or 8085 microprocessor (as is the Series II development system), refer to the 8086 FAMILY UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS.

Once the Bootstrap Loader has loaded your file, the Loader will transfer control to the start address of the main module in the file. When building your file, be absolutely certain that it includes only one main module. If it includes several main modules, the Bootstrap Operation is likely to fail. Typically, the start address of the main module is the entry point for the iRMX 86 initialization code.

Now that we have examined the requirements of the file to be loaded, let's examine the two options that can be used to select the file.

Loading A Default File

Unless you specifically configure the Bootstrap Loader to accept the file name from the terminal, it will load a default file. The name of the default file is /SYSTEM/RMX86. In other words, if you choose to use the default-file option, the Bootstrap Loader will start at the root directory of the selected device, find a subordinate directory named SYSTEM, and then find a data file named RMX86. The Bootstrap Loader will then load the data file.

If the default file does not exist on the bootstrap device, the Bootstrap Loader will halt.

Allowing the End User to Specify a File

You can configure the Bootstrap Loader to accept a file name from the terminal. If you elect to use this option, the Bootstrap Loader will prompt your end user for a file name. The Bootstrap Loader uses an asterisk (*) as a prompt character.

<u>Examples</u>. Be aware that this option can be used either with or without manual device selection. However, if you choose to use both this feature and manual device selection, your end users have several options:

- They can enter both a device name and a file name. For example,

  ```
  :F0:/wordprocessing
  ```

  This example causes the Bootstrap Loader to select device F0 and the named data file called wordprocessing located in the root directory of the device. Because the Bootstrap Loader does not distinguish between upper and lower case letters, "wordprocessing" could be replaced with "WORDPROCESSING" and the result would be the same.

- They can enter a device name and default the file name. For instance,

  ```
  :F0:
  ```

  This example causes the Bootstrap Loader to load file /SYSTEM/RMX86 from :F0:.

- They can enter a file name but default the device name. For example,

  ```
  /DATABASE
  ```

  This example causes the Bootstrap Loader to use automatic device selection. Once the device is selected, the Bootstrap Loader examines the root directory of the device, looking for the data file called "DATABASE".

- They can default both the device name and the file name by entering only a carriage return. This will cause the Bootstrap Loader to use automatic device selection and to load from the file named /SYSTEM/RMX86.

Syntax of File Names. The syntax of the file name is, with one exception, identical to the syntax of a subpath in the Basic I/O System. The exception relates to the up-arrow (↑) character or, as it appears on some terminals, the circumflex (^) character. The Bootstrap Loader deems invalid any file name containing this character.

Interpretation of File Names. With one exception, the Bootstrap Loader interprets the file name in the same manner that the Basic I/O System interprets subpath parameters for named files. The one exception occurs when the file name begins with a character other than a slash (/).

If your end user enters a file name that does not begin with a slash, the Bootstrap Loader will place /SYSTEM/ at the front of the file name provided by your end user. For example, if your end user enters the name

    DATABASE

the Bootstrap Loader will behave as though the end user had entered

    /SYSTEM/DATABASE

This rule also applies when your end user enters both a device name and a file name. For instance, if the end user enters

    :F0:wordprocessing

the Bootstrap Loader will behave as though the end user had entered

    :F0:/SYSTEM/wordprocessing

Interpretation of Combinations of Devices and Files. If you configure the Bootstrap Loader to use manual device selection and to allow the end user to select the file to be loaded, the following rules govern the interpretation of the information entered by the end user:

- The Bootstrap Loader examines the first character entered. If it is a colon (:) the Bootstrap Loader attempts to parse a device name. Once it has the device name, it attempts to find the device in your device table. If it is unable to find the device, it changes from manual device selection to automatic device selection, and it reprocesses all of the information (including the colon) as though it were simply a file name.

- If the first character entered is not a colon, the Bootstrap Loader switches to automatic device selection and attempts to interpret as a file name all of the information entered through the terminal.

Processing Accorded Invalid File Names. If your end user enters a file name that is invalid (for instance, one containing an up-arrow), the Bootstrap Loader will halt.

Processing Accorded Files Not on the Device. If the Bootstrap Loader is not able to find the file on the bootstrap device, the Bootstrap Loader will halt.

PUTTING THE SECOND STAGE ON THE VOLUME

Because second stage is read from the bootstrap device into RAM, the second stage must somehow be placed on the volume contained by the bootstrap device. If you are using Release 2 or a more recent version of the iRMX 86 Operating System, this placement occurs without any special effort on your part. Whenever you format a volume for use with the iRMX 86 Operating System, the formatting process will place the second stage on the volume.

INVOKING THE BOOTSTRAP LOADER

There are two ways to invoke the Bootstrap Loader. They are:

- automatic invocation upon reset
- invocation under program control

You can provide your user with either or both methods depending upon your actions during configuration of your system. The following paragraphs describe the methods of invocation and the actions that you must take to provide your end user with each method.

AUTOMATIC INVOCATION UPON RESET

If you choose to provide automatic invocation of the Bootstrap Loader, you must use the BOOTSTRAP switch of the LOC86 command when you locate the first stage. For details regarding the LOC86 command, refer one of the following manuals:

- If you are using a development system that incorporates an iAPX 86 microprocessor (for example, a Series III development system), refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS.
- If you are using a development system that incorporates only an 8080 or an 8085 microprocessor (for example, a Series II development system), refer to the 8086 FAMILY UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS.

From the point of view of your end user, automatic invocation means that whenever an operator or the software triggers the RESET signal for the iAPX 86 microprocessor, the Bootstrap Loader will start running. For instance, if you build a RESET button into your hardware and wire it to the RESET signal of the iAPX 86, your end user can invoke the Bootstrap Loader by pressing the RESET button.

INVOCATION UNDER PROGRAM CONTROL

You can invoke the Bootstrap Loader from software. To do this, you need only jump to the entry point of the Bootstrap Loader which is called BOOTSTRAP (a PUBLIC symbol). In other words, just code a jump to BOOTSTRAP and define BOOTSTRAP as an EXTERNAL symbol in your module. Later, when linking your application software, be sure to link the first stage of the Bootstrap Loader. To find the name of the file containing the first stage of the Bootstrap Loader, refer to the iRMX 86 CONFIGURATION GUIDE.

HOW THE FIRST STAGE COMMUNICATES WITH A TERMINAL

If you configure the first stage of the Bootstrap Loader to use manual device selection or end-user file selection, you must include a terminal in your system. Furthermore, because the Terminal Handler is generally not in memory when the Bootstrap Loader runs, you must add software to allow the Bootstrap Loader to communicate with the terminal.

There are three ways in which you can provide this software:

- First, you can use the CO (console output) and CI (console input) procedures provided by the iSBC 957A package. This option is only feasible if your system will always incorporate the iSBC 957A package.
- Second, you can use the Intel-provided source code for CO and CI. This source code is provided as part of the iRMX 86 product.
- Third, you can write your own CO and CI procedures.

For guidance in implementing your choice, refer to the Bootstrap Loader chapter of the iRMX 86 CONFIGURATION GUIDE.

ERROR PROCESSING

Some systems using the Bootstrap Loader do not include a terminal. Consequently, the Loader does not display error messages when it encounters a problem that prevents it from successfully loading your software.

Even so, by noting the behavior of the Bootstrap Loader when it fails to successfully load, you can find out the cause of failure and take steps to eliminate it. Table 3-1 shows the correlation between the behavior of the Bootstrap Loader and the possible causes of its failure.

TABLE 3-1. Postmortum Analysis of Bootstrap Loader Failure

BEHAVIOR OF LOADER	POSSIBLE CAUSES
Bootstrap Loader halts in first stage.	If you are using no device selection, your device is not ready. An I/O error occurred during the loading operation.
Bootstrap Loader halts in second stage.	The syntax of the file or device name is incorrect. A checksum error occurred during the loading operation. The Bootstrap Loader was not able to find the specified file on the bootstrap device.
Bootstrap Loader loops in first stage.	You have configured the Bootstrap Loader to use automatic or manual device selection, but you have not readied the device.
Bootstrap Loader loops in second stage.	The Bootstrap Loader is attempting to load the system on top of the second stage. The Bootstrap Loader is attempting to load the system into nonexistent memory.

CHAPTER 4. DEVICE DRIVERS FOR THE BOOTSTRAP LOADER

As discussed in Chapter 3, the iRMX 86 Bootstrap Loader can be configured to run with many kinds of devices. If you wish to use one of the devices for which Intel supplies a device driver, you do not need to read this chapter.

On the other hand, if you wish to use the Bootstrap Loader with a device other than those supported by Intel, you must write your own device driver. The purpose of this chapter is to provide you with guidelines for writing a customized driver.

A device driver for the Bootstrap Loader must consist of two procedures. The Bootstrap Loader calls one of these, the initialization procedure, to initialize the bootstrap device before the Loader begins reading from the device. The Bootstrap Loader calls the other procedure, the reading procedure, to load information from the device.

For simplified notation, the remainder of this chapter refers to the two procedures as DEVICE$INIT, and DEVICE$READ. However, you can actually provide them with any name you wish during the process of configuring the system.

Both of the procedures must obey the Large model of the PL/M-86 programming language. This means that the procedures must be FAR (as opposed to NEAR) and all pointers must be 32 bits. For more information regarding the PL/M-86 Large model of computation, refer to one of the following two manuals:

- If your development system includes an iAPX 86 microprocessor (as in the Series III), refer to the PL/M-86 USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS.

- If your development system does not include an iAPX 86 microprocessor (as in the Series II), refer to the PL/M-86 COMPILER OPERATING INSTRUCTIONS FOR 8080/8085-BASED DEVELOPMENT SYSTEMS.

Be aware that you can write the procedures in assembly language. But if you do, you must adhere to the requirements of a PL/M-86 Large procedure.

DEVICE$INIT PROCEDURE

The DEVICE$INIT procedure must present the following PL/M-86 interface to the Bootstrap Loader:

```
DEVICE$INIT:   PROCEDURE (UNIT) WORD;

               DECLARE UNIT WORD;
```

where UNIT is the device's unit number as defined during configuration of the Bootstrap Loader. Refer to the iRMX 86 CONFIGURATION GUIDE for more information about device unit information.

The WORD value that is returned by the procedure must be the device granularity in bytes.

The following outline shows the steps that the DEVICE$INIT procedure must perform to be compatible with the Bootstrap Loader:

1) Test to see if the device is present. If it is not, return the value zero.

2) Initialize the device for reading. This is a highly 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 with a value of zero. If initialization was successful, continue on to Step 4.

4) Obtain the device granularity. For some devices, only one granularity is possible while, for others, several granularities are possible. This is a device-dependent issue that is explained in the hardware reference manual for your device.

5) Return to the caller with the device granularity.

DEVICE$READ PROCEDURE

The DEVICE$READ procedure must present the following PL/M-86 interface to the Bootstrap Loader:

```
DEVICE$READ:   PROCEDURE (UNIT, BLK$NUM$HI, BLK$NUM$LO, BUF$PTR);

               DECLARE   UNIT        WORD,
                         BLK$NUM$HI  WORD,
                         BLK$NUM$LO  WORD,
                         BUF$PTR     POINTER;
```

where

UNIT — is the device-unit number as defined during the process of configuring the Bootstrap Loader. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.

BLKNUMHI — is a 16-bit number that provides the Bootstrap Loader with the most significant 16 bits of the number of the block to be read.

BLKNUMLO — is a 16-bit number that provides the Bootstrap Loader with the least significant 16 bits of the number of the block to be read.

BUF$PTR — is a 32-bit POINTER to the buffer that is to receive the information from the secondary storage device.

The DEVICE$READ procedure does not return a value to the caller.

The following outline shows the steps that the DEVICE$READ procedure must perform to be compatible with the Bootstrap Loader:

1. Read the block specified by the BLK$NUM parameters from the bootstrap device specified by the UNIT parameter into the memory location specified by the BUF$PTR parameter.

2. Check for I/O errors. If one occurred, halt. Otherwise, return to the caller.

CHPATER 5. USING THE APPLICATION LOADER

The Application Loader is a powerful tool that provides your tasks with the means of loading code from secondary storage into RAM. This chapter is designed to help you understand the capabilities of the Loader by providing you with background information. The chapter consists of four major parts:

- Loader Terminology
- Loader Features
- Configuration Options
- Preparing Code for Loading

After reading this chapter and Chapter 6, you should be able to understand the system calls in Chapter 7.

LOADER TERMINOLOGY

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

- object code
- object module
- object file
- synchronous system call
- asynchronous system call
- absolute code
- position-independent code (PIC)
- load-time locatable code (LTL)
- fixup
- I/O job
- overlay
- root module
- overlay module

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

OBJECT CODE

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

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

OBJECT MODULE

An object module is the output of a single compilation, a single assembly, or a single invocation of the LINK86 or LOC86 commands.

OBJECT FILE

An object file is a named file in secondary storage. The file contains object code in one or more modules.

SYNCHRONOUS SYSTEM CALL

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

ASYNCHRONOUS SYSTEM CALL

An asynchronous system call is one in which the calling task can run concurrently with the invoked system call. For a detailed explanation of the behavior of asynchronous system calls, read Chapter 6.

ABSOLUTE CODE

Absolute code is one of three forms in which object code can appear. An absolute object module is one that has been processed by LOC86 to run only at a specific location in memory. Consequently, the Application Loader loads an absolute object module only into the specific location that the module must occupy.

POSITION INDEPENDENT CODE (PIC)

Position independent code (commonly referred to as PIC) is one of three forms in which object code can appear. PIC differs from absolute code in that PIC can be loaded into any memory location. Consequently, when the Application Loader is requested to load PIC, the Loader obtains iRMX 86 segments and loads the PIC into the segments.

The advantage of PIC over absolute code is that PIC does not require you to reserve a specific block of memory. When the loading operation begins, the Loader obtains memory from the pool of the job in which the loader runs.

LOAD-TIME LOCATABLE (LTL) CODE

Load-time locatable code (commonly referred to as LTL code) is the third form in which object code can appear. LTL code is similar to PIC in that LTL code can be loaded anywhere in memory. The difference is that LTL code can be used by tasks having more than one code segment or more than one data segment. In contrast, PIC is restricted to tasks having one code segment and one data segment.

The techniques used to generate absolute, PIC, and LTL code are discussed later in this chapter.

FIXUP

When the Application Loader loads an LTL program, the Loader must adjust some of the code. This adjustment is known as a fixup, or more accurately as a base fixup.

The reason for this adjustment is that the pointers used in the LTL code must be independent of the contents of the registers in the microprocessor. While loading the LTL code, the Application Loader changes the base portion of the pointers as needed, providing this independence. Without this adjustment, the Loader could not support tasks having multiple code segments or data segments.

I/O JOB

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

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

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

OVERLAY

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

- As a monolithic program that occupies a large amount of RAM whenever the compiler is running.
- As an overlayed structure in which pieces of the compiler reside in RAM only when they are being used.

As a monolithic program the compiler can reside on secondary storage until it is needed, but once needed, the entire collection of object code must be loaded into RAM. In contrast, as an overlayed program, the pieces (called overlays) of the compiler all reside on secondary storage, and individual overlays are loaded as they are needed.

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

ROOT MODULE AND OVERLAY MODULES

A root module is an object module that controls the loading of overlays. Let's again use an overlayed compiler as an example. Suppose that you are developing an application system on which your customers will compile programs. When your customer invokes the compiler, your application system should use one of the A$LOAD, A$LOADIOJOB, or S$LOAD$IO$JOB system calls to load the root module of the compiler. The root module should then use the S$OVERLAY system call to load the overlay modules as they are needed.

For more information regarding the notion of overlays, root module, and overlay module, refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS.

LOADER FEATURES

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

- Device Independence
- Ability to Load Three Kinds of Code
- Synchronous and Asynchronous System Calls
- Support for Overlayed Programs
- Configurability

The following sections briefly discuss each of these features.

DEVICE INDEPENDENCE

The Application Loader can load object code from any device that supports iRMX 86 named files. Your iRMX 86 Operating System is delivered with device drivers that support named files on any of the following devices:

- iSBC 204 Single Density Flexible Disk Controller
- iSBC 206 Hard Disk Controller
- iSBC 215 Winchester Hard Disk Controller
- iSBC 218 Multimodule Flexible Disk Controller
- iSBC 220 SMD Hard Disk Controller
- iSBC 254 Bubble Memory Board

Furthermore, if you wish to load from a device for which Intel does not yet supply a device driver, you can write your own device driver. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM for directions.

ABILITY TO LOAD THREE KINDS OF CODE

The iRMX 86 Application Loader can load absolute code, PIC, and LTL code.

SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

The Application Loader provides you with both synchronous system calls and asynchronous system calls. If you want your tasks to explicitly control the overlapping of processing with loading operations, you can use asynchronous system calls. On the other hand, if you prefer ease of use to explicit control, you can use synchronous system calls.

SUPPORT FOR OVERLAYED PROGRAMS

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

CONFIGURABILITY

The Application Loader is configurable. During the process of configuring the rest of the iRMX 86 Operating System, you can select the features of the Application Loader that your application system needs. If you don't need all of the capabilities of the Loader, you can leave out some options and use a smaller, faster version of the Loader.

CONFIGURATION OPTIONS

The Application Loader has two kinds of configuration options. You can specify the kind of code you wish to be able to load, and you can specify the system calls that you want included in your application.

The following sections discuss the options from which you can choose when you configure the Loader, but they do not tell you how to specify your choices. To find out how to specify your choices, refer to the iRMX 86 CONFIGURATION GUIDE.

CHOICE OF LOADING CAPABILITIES

During configuration, you can provide your application system with the ability to perform any of the following loading operations:

- The ability to load absolute code only. This is the smallest and fastest option. So, if memory and performance considerations are more important to you than the ability to load code into iRMX 86 segments, you should consider this option.

- The ability to load both PIC and absolute code. This option provides you with a larger Loader than does the absolute-only option. However, this is the smallest configuration that enables the Loader to create iRMX 86 segments into which it can load your code.

- The ability to load LTL code, PIC, and absolute code. This combination is only slightly larger than the PIC-and-absolute-code configuration, but it provides the ability to load code for tasks that require more than one code segment or more than one data segment.

- The ability to load overlays, LTL code, PIC, and absolute. This is the most powerful Loader configuration in that it provides your application system with the ability to perform all loading operations. This option provides the S$OVERLAY system call.

CHOICE OF LOADING METHODS

While configuring the Loader, you can select one of the following loading methods:

- The ability to load code without creating an I/O job. This option provides the A$LOAD system call.

- The ability to load code asynchronously with or without creating an I/O job. This option provides both the A$LOAD and the A$LOADIOJOB system calls.

- The ability to load code synchronously or asynchronously by creating an I/O job, and the ability to load code asynchronously without creating an I/O job. This option provides the A$LOAD, A$LOADIOJOB, and S$LOAD$IO$JOB system calls.

PREPARING CODE FOR LOADING

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

- The kind of development system you are using.

- The PL/M-86 model of computation to which you are adhering.

- Whether or not you want the loaded calls to be able to invoke iRMX 86 system calls.

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

DEVELOPMENT SYSTEMS

Because you use a development system to prepare the object code that you plan to load, the kind of development system(s) that you use significantly affects your capabilities. Be aware of the following two facts:

- If you use a development system that is based on an 8080 or 8085 microprocessor (a Series II, for example), you can only generate absolute code for loading. You cannot successfully generate PIC or LTL code. Furthermore, you cannot use the S$OVERLAY system call provided by the Application Loader.

  In contrast, if you use a development system that is based on an iAPX 86 or an iAPX 88 microprocessor (Series III for example), you can generate any of the three kinds of object code, and you can use the S$OVERLAY system call.

- You should not use a combination of development systems to generate object files. For example, suppose that you compile your source file using a Series II development system, and then you link the code using a Series III. The utilities on the two systems do not generate identical object records. Consequently, there is a reasonable chance that the resultant object code can not be loaded by the Application Loader.

  For the balance of this chapter, all information is based on the assumption that you are compiling, linking, and locating on one kind of development system.

PL/M-86 MODELS OF COMPUTATION

When you compile your source code, you must (explicitly or implicitly) select a PL/M-86 model of computation. (This is also known as a size control.) The model you select can greatly affect the kind of object code generated. The purpose of this section is to correlate the model of computation with the kind of code generated.

The PL/M-86 programming language provides four models of computation. They are SMALL, MEDIUM, LARGE, and COMPACT. For more information regarding these models and their effect on the iRMX 86 Operating System, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.

When you compile the code that you plan to load, you specify (either explicitly or by default) one of these four models. The following three sections explain the effect of your choice on the object code.

PL/M-86 Small Model

The iRMX 86 Operating System does not support the PL/M-86 SMALL model of computation. Do not use it to generate any code that you plan to load with the Application Loader.

PL/M-86 Medium and Large Models

If you use the MEDIUM or LARGE model of PL/M-86, you cannot generate PIC. This means that you must choose between absolute code and LTL code.

If you use both LINK86 and LOC86 to process the output of the PL/M-86 compiler, you will obtain absolute code. This is true for both 8080/8085-based development systems and for 8086-based development systems.

If you use only LINK86 (with the BIND control) to process the output of the compiler, you will obtain LTL code. This is true only for 8086-based development systems. If your system is an 8080/8085-based development system, you cannot load the output generated by the LINK86 command.

PL/M-86 Compact Model

If you use the COMPACT model of PL/M-86, and if you have an 8086-based development system, you generate absolute code, PIC, and LTL code. On the other hand, if you have an 8080/8085-based development system, you can generate only absolute code.

If you use both LINK86 and LOC86 to process the output of the PL/M-86 compiler, you will obtain absolute code. This is true for both 8080/8085-based development systems and for 8086-based development systems.

If you use only LINK86 (with the BIND control) to process the output of the compiler, you can obtain PIC by adhering to the following guidelines when creating your source code:

- Do not use an INITIAL statement or a DATA statement to initialize a POINTER.
- Do not use the INTVEC control for any interrupt procedures. Be aware that INTVEC is the default control. This means that you must invoke the NOINTVEC control for any interrupt procedures.

Failure to adhere to these guidelines will cause the object module to be LTL code rather than PIC.

INVOKING iRMX 86 SYSTEM CALLS

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

ENTRY POINTS AND STACK SIZES

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

There is one circumstance in which the Loader does not require an entry point. If your task implicitly knows the entry point (for instance, if the entry point is at a reserved location in memory) and if your task uses the A$LOAD system call to load the code, then the object file need not specify an entry point.

You can use either of the following techniques to ensure that your object file specifies an entry point:

- Write your source code as a main module. This will automatically ensure that the object module contains a start address. You can use this technique for absolute code, PIC, or LTL code.

- Write your source code as a procedure rather than as a main module. Later, when using LOC86 to convert your object code to absolute code, use the START control to designate the entry point. Because this technique requires that you use LOC86, you cannot use this technique for PIC or LTL code.

The following two sections provide more information about these techniques.

Using a Main Module

If you are loading PIC or LTL code, you must write your source code as a main module. If you are loading absolute code that was generated on an 8086-based development system (such as Series III) you should write your source code as a main module. And, if you are loading code generated on an 8080/8085-based development system, you can write your source code as a main module. To prepare your main module, perform the following two steps:

1) When linking (using the LINK86 command) or locating (using the LOC86 command) your code, you must use the SEGSIZE(STACK(...)) control of the command to assign an appropiate stack size. If you are using an 8086-based development system, you can assign the stack size with either the LINK86 or LOC86 command. However, if you are using an 8080/8085-based development system, you can assign the stack size only with the LOC86 command. You can find a description for this control in the iAPX 86,88 FAMILY USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS or in the 8086 FAMILY UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS. To find out how much stack to assign, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.

2) If you are using the A$LOAD system call and you plan to run the loaded code as a task, you must take one of the following courses of action:

   - If you are loading PIC, LTL code, or if you are loading absolute code that was generated on an 8086-based development system with the NOINITCODE control of the LOC86 command, then the Loader will tell the calling task what parameters to use when invoking the CREATE$TASK or CREATE$JOB system call. These parameters include the entry point and the stack size for the new task. The Loader uses the Loader Result Segment to return this information to the calling task. Refer to the description of the A$LOAD system call in Chapter 7 for more information.

   - If your object code is absolute code that was created on an 8080/8085-based development system, or if it is absolute code that was created on an 8086-based system without the NOINITCODE control of the LOC86 command, you must allow the iRMX 86 Nucleus to create a stack for you. To do this, you must specify a 0:0 for the stack pointer parameter of the CREATE$TASK or the CREATE$JOB system call.

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

     In order to minimize the amount of memory wasted by this stack switching operation, you should specify a small stack size (128 decimal bytes) in the CREATE$TASK system call or the CREATE$JOB system call. This Nucleus-allocated stack need not be large because it is only used if the task is interrupted before it switches stacks.

Be aware that the stack switching technique has one less-than-desirable side effect. If you use the iRMX 86 Debugger, it will always indicate that the stack for the loaded code has overflowed. This overflow indication is caused by the main module switching stacks, rather than by an actual overflow. Although you can generally ignore this overflow indication, you should be aware that a real overlow can occur and, if it does, the Debugger can not advise you of it. If you find this side effect to be unacceptable, you can eliminate it by writing your source code as a procedure.

For more information about the CREATE$TASK or the CREATE$JOB system calls refer to the iRMX 86 NUCLEUS REFERENCE MANUAL. For information about the iRMX 86 Debugger, refer to the iRMX 86 DEBUGGER REFERENCE MANUAL.

Using a Procedure

It is to your advantage to write your source code as a procedure only if the following three statements are true:

- You are loading absolute object code that was generated on an 8086-based development system without using the NOINITCODE control of the LOC86 command, or you are loading absolute object code that was generated on an 8080/8085-based development system.

- You are using the A$LOAD system call to load the code.

- You are going to run the object code as a task after the loading operation is completed.

If any of these statements is not true, you should write the source code as a main module rather than as a procedure.

The process of loading a procedure is more restrictive than that of loading a main module, but it does have one advantage. You can avoid the stack-switching side effect. In other words, you can load absolute code and create a task without losing the Debugger's ability to detect stack overflow. You also avoid wasting memory by avoiding stack switching.

In order to successfully load code that is written as a procedure, you must adhere to the following four rules:

- The code must adhere to the PL/M-86 LARGE model of computation. This means that you must either compile the procedure using the LARGE control, or you must follow the calling conventions of the LARGE model. Refer to one of the following manuals for information about the PL/M-86 LARGE model of computation:

    - PL/M-86 USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS

    - PL/M-86 COMPILER OPERATING INSTRUCTIONS FOR 8080/8085-BASED DEVELOPMENT SYSTEMS

- When you invoke the LOC86 command to assign absolute addresses to your object code, use the START control to select one of the PUBLIC symbols in your procedure as an entry point. Also specify SEGSIZE(STACK(0)) to set the stack to zero length. For more information about the START and SEGSIZE controls, refer to one of the following manuals:

  - iAPX 86,88 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS

  - 8086 FAMILY UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS

- When you invoke the CREATE$TASK system call or the CREATE$JOB system call, allow the Nucleus to dynamically allocate a stack for the new task. Do this by setting the stack pointer parameter to 0:0. However, be certain that you specify a stack size parameter that is large enough to accommodate the task. For guidelines in determining stack sizes, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.

- When you invoke the CREATE$TASK system call or the CREATE$JOB system call, set the data segment base parameter to 0. The reason for this is that a procedure adhering to the LARGE model of computation always initializes its own data segment.

CHAPTER 6. ASYNCHRONOUS SYSTEM CALLS

Each asynchronous system call has two parts -- one sequential, and one concurrent. As you read the descriptions of the two parts, refer to Figure 6-1 to see how the parts relate.

- the sequential part

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

- the concurrent part

  The concurrent part runs as an iRMX 86 task. The task is made ready by the sequential part of the call, and it runs only when the priority-based scheduling of the iRMX 86 Operating System gives it control of the processor. The concurrent part also returns a condition code.

The reason for splitting the asynchronous calls into two parts is performance. The functions performed by these calls are somewhat time-consuming because they involve mechanical devices such as disk drives. By performing these functions concurrently with other work, the Application Loader allows your application to run while the Loader waits for the mechanical devices to respond to your application's request.

Let's look at a brief example showing how your application can use asynchronous calls. Suppose your application must load a program that is stored on disk. The application issues the A$LOAD system call to have the Application Loader load the program into memory. Let's trace the action one step at a time:

1. Your application issues the A$LOAD system call. This call requires, as do all asynchronous calls, that your application specify a response mailbox for communication with the concurrent part of the system call.

2. The sequential part of the A$LOAD call begins to run. This part checks the parameters for validity.

3. If the sequential part of the call detects a problem, it places a sequential exception code in the word to which your except$ptr parameter points. It then returns control to your application. It does not make ready the Application Loader task to perform the loading function.

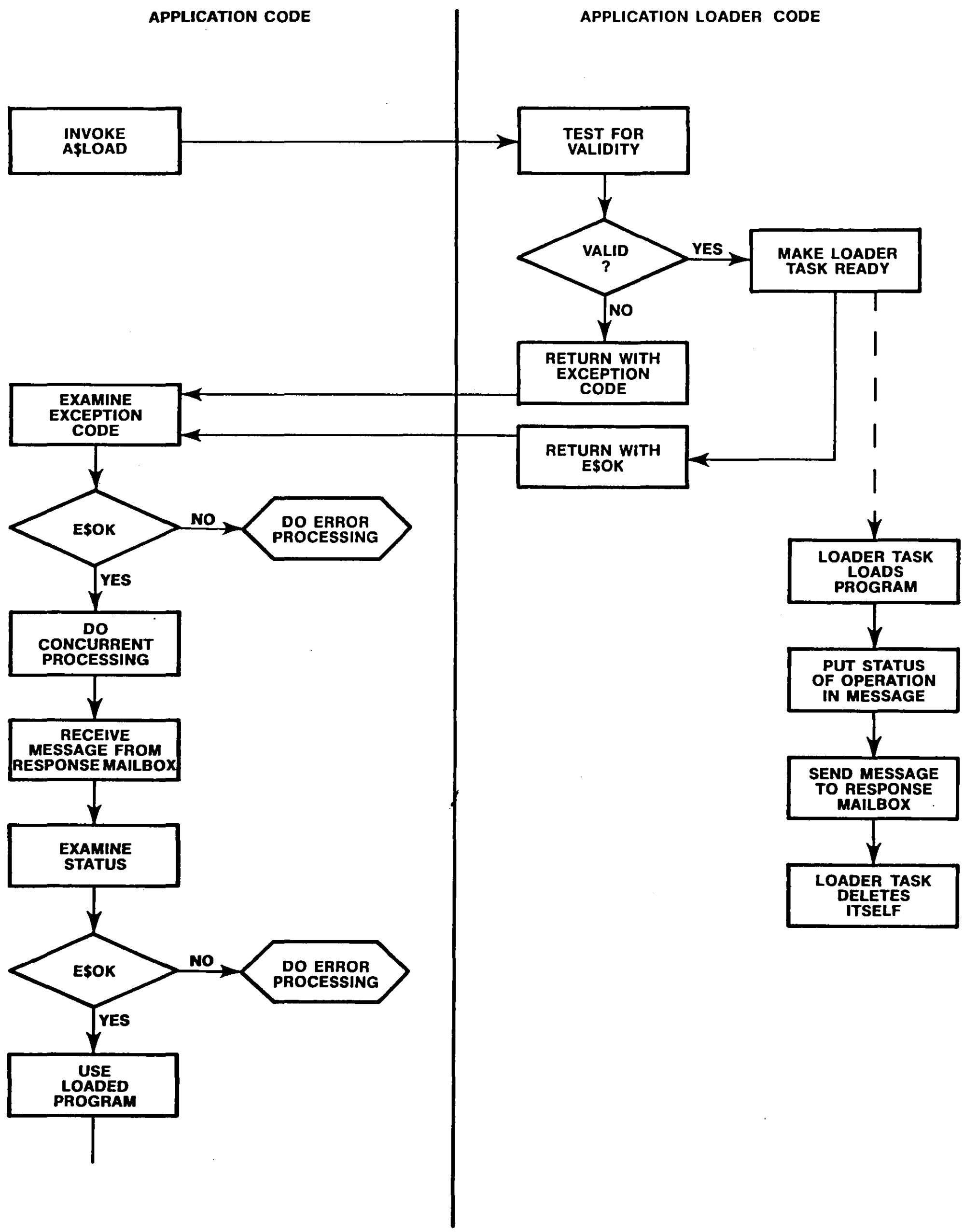

Figure 6-1. Concurrent Behavior of an Asynchronous System Call

4. Your application receives control. Its behavior at this point depends on the condition code returned (to the word specified by the except$ptr parameter) by the sequential part of the system call. Therefore, the application tests the sequential condition code. If the code is E$OK, the application continues running until it must use the program loaded from the disk. It is at this point that your application can take advantage of the asynchronous and concurrent behavior of the Application Loader.

   For example, your application can use this overlapping processing to perform computations. The point is that you can decide what you want your application to do while the asynchronous system call is running.

   On the other hand, if your application finds that the sequential condition code is other than E$OK, the application can assume that the Application Loader did not make ready a task to perform the function.

   For the balance of this example, we will assume that the sequential part of the system call returned an E$OK sequential condition code.

5. Your application now must use the loaded program. Before doing so, your application must verify that the concurrent part of the A$LOAD system call ran successfully. The application issues a RECEIVE$MESSAGE system call to check the response mailbox that the application specified when it invoked the A$LOAD system call.

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

In the foregoing example, we used a specific system call (A$LOAD) to show how asynchronous calls allow your application to run concurrently with loading operations. Now let's look at some generalities about asynchronous calls.

- All of the asynchronous system calls consist of two parts -- one sequential and one concurrent. The Application Loader will activate the concurrent part only if the sequential part runs successfully (returns E$OK).

- Every asynchronous system call requires that your application designate a response mailbox for communication with the concurrent part of the system call.

- Whenever the sequential part of an asynchronous system call returns a condition code <u>other than</u> E$OK, your application should not attempt to receive a message from the response mailbox. There can be no message because the Application Loader cannot run the concurrent part of the system call.

- Whenever the sequential part of an asynchronous system call runs successfully (E$OK), your application can count on the Application Loader running the concurrent part of the system call. Your application can take advantage of the concurrency by doing some processing before receiving the message from the response mailbox.

- Whenever the concurrent part of a system call runs, the Application Loader signals its completion by sending an object to the response mailbox. The precise nature of the object depends upon which system call your application invoked. You can find out what kind of object comes back from a particular system call by looking up the call in Chapter 7 of this manual.

- Whenever the Application Loader returns a segment to your application's response mailbox, your application must delete the segment when it is no longer needed. The Application Loader uses memory for such segments, so if your application fails to delete the segment, your application system may run short of memory.

CHAPTER 7. SYSTEM CALLS OF THE APPLICATION LOADER

This chapter describes the PL/M-86 calling sequences for the system calls provided by the Application Loader. In this chapter, the system calls are listed alphabetically according to the same shorthand notation used throughout this manual. For example, A$LOAD precedes A$LOADIOJOB. This shorthand notation is language independent and should not be confused with the actual form of the PL/M-86 call. The precise format of each call is spelled out as part of the detailed description.

Be aware that iRMX 86 system calls are declared as typed or untyped external procedures in the PL/M-86 language. When you write a program in PL/M-86, you can use these procedures to invoke the system calls provided by the Application Loader.

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

RESPONSE MAILBOX PARAMETER

Two of the system calls described in this chapter are asynchronous. These are the A$LOAD and the A$LOADIOJOB system calls.

As explained in Chapter 6, your task must specify a mailbox whenever the task invokes an asynchronous system call. The purpose of this mailbox is to receive information describing the result of the asynchronous operation.

When you examine the detailed descriptions of the A$LOAD and the A$LOADIOJOB system calls, you will find a parameter called response$mbox. Your tasks must use this parameter to tell the Application Loader where to send the Loader Result Segment that describes the outcome of the operation.

The format of the Loader Result Segment depends upon which system call was invoked. Consequently, this manual describes the format of the Loader Result Segment in the detailed descriptions of the system calls.

You must be aware of a potential problem associated with the use of response mailboxes. If your task uses the same response mailboxes for several invocations of asynchronous system calls, it is possible for the Application Loader to return the Loader Result Segments in an order different than the order of invocation. Your tasks can avoid this problem by using a different mailbox for each invocation of an asynchronous system call.

CONDITION CODES

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

CONDITION CODES FOR SYNCHRONOUS SYSTEM CALLS

For those system calls that are strictly synchronous (S$LOAD$IO$JOB and S$OVERLAY), the Application Loader returns only one condition code. Your task can deal with this code by using the techniques described in the iRMX 86 NUCLEUS REFERENCE MANUAL.

CONDITION CODES FOR ASYNCHRONOUS SYSTEM CALLS

For the system calls that are asynchronous (A$LOAD and A$LOADIOJOB), the Application Loader can return two condition codes. One code is returned after the sequential part of the system call is executed, and the other is returned after the concurrent part of the call is executed. (Refer to Chapter 6 for a discussion of the sequential and concurrent parts of an asynchronous system call.) Your task must process each of these two condition codes in a different manner.

Sequential Condition Codes

The Application Loader returns the sequential condition code in the WORD indicated by the except$ptr parameter. Your task can deal with this condition code by using the techniques described in the iRMX 86 NUCLEUS REFERENCE MANUAL.

Concurrent Condition Codes

The Application Loader returns the concurrent condition code in the Loader Result Segment that it sends to the response mailbox. Your task must explicitly examine this condition code. If the code is E$OK, the asynchronous loading operation ran successfully. If the code is other than E$OK, a problem occurred during the asynchronous loading operation, and your task must decide what to do about the problem. Your task cannot use the techniques provided by the Nucleus for processing the concurrent condition code.

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

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

INPUT PARAMETERS

connection — is a WORD containing a token for a connection to the file that the Application Loader will load. The connection must satisfy four requirements:

- It must have been created in the calling task's job.
- It must be a connection for a named file.
- It must have READ access to the file.
- It must be closed.

If the connection does not satisfy all four of these requirements, the Application Loader will return an exceptional condition.

response$mbox — is a WORD containing a token for the mailbox to which the Application Loader will send the Loader Result Segment after the concurrent part of the system call has finished running. The format of the Loader Result Segment is described later in this description.

OUTPUT PARAMETERS

except$ptr — is a POINTER to a WORD in which the Application Loader will place the condition code generated by the sequential part of the system call.

A$LOAD (continued)

DESCRIPTION

The purpose of this system call is to allow your task to load programs from secondary storage into main memory. This system call does not cause the program to be made part of a job or a task, nor does it place the program into execution. If you wish to execute the program or make it part of a task or a job, the calling task must explicitly do this.

Asynchronous Behavior

This system call is asynchronous. It allows the calling task to continue running while the loading operation is in progress. When the loading operation is finished, the Application Loader will send a Loader Result Segment to the mailbox designated by the response$mbox parameter. Refer to Chapter 6 for a detailed description of asynchronous behavior.

File Sharing

The Application Loader does not expect exclusive access to the file. However, if another task is also using the file, the file sharing must obey the following two guidelines:

- Other tasks can use the file only through other connections. Your task should not attempt to share the connection passed to the Application Loader.

- If other tasks use the file, they should use it only for reading. The Application Loader marks the file as being sharable with readers only.

Considerations Relating to Code Type

If the file being loaded is absolute code, the Loader will not create segments to accept the code. Rather, it will simply load the program into the memory locations that the object file is designed to occupy. To exclude the possibility of loading over existing information, you should avoid including these memory locations in any memory pools. Refer to the iRMX 86 CONFIGURATION GUIDE to see how to reserve memory locations. Also, you must ensure that the code is not loaded over the Operating System.

In contrast, if the file being loaded is position-independent code (PIC) or load-time locatable (LTL) code, the Application Loader will create the segments required to contain the loaded program. Be aware that, once your task no longer needs the loaded program, your task should delete these segments. The Application Loader does not delete them.

DESCRIPTION (continued)

Effects of Model of Computation

If the program being loaded adheres to the PL/M-86 COMPACT model of computation, the Application Loader will return (in the Loader Result Segment) tokens for the stack, code, and data segments. On the other hand, if the program adheres to the LARGE or MEDIUM models of computation, the Application Loader cannot return this information.

This means that, if the program is LARGE or MEDIUM, the calling task cannot know the location of the loaded program's stack, code or data segments. Consequently, the calling task cannot delete any of the stack, data or code segments.

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

Deleting Loader Result Segments

The Application Loader uses memory from the pool of the calling task's job to create the Loader Result Segment for this system call. If the calling task does not delete the segment after it is no longer needed, the segment will occupy memory that could be used for other purposes. In fact, if the calling task performs repeated loading operations, failure to delete the Loader Result Segments could lead to E$MEM exception codes.

Format of the Loader Result Segment

The Loader Result Segment has the following form:

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

A$LOAD (continued)

DESCRIPTION (continued)

where:

except$code — This is the condition code for the concurrent part of the system call. If the value is other than E$OK, some problem occurred during the loading operation.

record$count — This contains the number of object records read by the Application Loader on behalf of this invocation of the A$LOAD system call. If the except$code indicates that the loading operation terminated before completion, record$count contains the number of the last record that was read.

errorrectype — This identifies the type of record that caused the loading operation to fail. If the loading operation is successful (that is, if except$code is E$OK), this BYTE will be zero.

num$undefined$refs — An external fixup usually (but not always) indicates an error during the linking process. The Loader will continue to run even if an object file contains external fixups. The purpose of this num$undefined$refs depends upon the kind of code that your Application Loader is configured to load:

- If the Loader is configured for LTL code, this WORD contains the number of external fixups that the Loader detected during the loading operation.

- If the Loader is configured to load PIC or absolute code, the Loader will set this WORD to 1 or 0. If the Loader found no external fixups during the loading operation, this WORD will be set to 0. If external fixups were found, this WORD will be set to 1.

init$ip — This WORD contains the initial value for the loaded program's instruction pointer (IP register). The calling task can use this information in either of two ways:

- It can use it to invoke the CREATE$TASK system call.

- It can jump to this location within the code segment of the loaded program.

DESCRIPTION
init$ip (continued)

The Loader sets this variable to zero if the file does not specify an initial value for the instruction pointer. This can only happen when the file contains only procedures and no main module.

codesegbase — This is the base for the code segment that contains the entry point for the loaded code. If the loaded program does not contain a main module, the Loader cannot ascertain this information, so the Loader will place a value of zero in this variable.

Be aware that codesegbase can be used in conjunction with init$ip as a POINTER to the entry point of the loaded program.

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

The Loader sets this variable to zero under two circumstances. First, if there is no main module, the object file does not specify the stack offset, and the Loader will set this variable to zero. Second, if you have a main module, but the Loader still sets this variable and the stacksegbase to zero anyway, then the loaded code dynamically initializes the SP and SS registers.

stacksegbase — This is the base for the stack segment for the loaded program. The calling task can use this value to initialize the SS (stack segment) register.

The Loader sets this variable to zero under two circumstances. First, if there is no main module, the object file does not specify the stack base, and the Loader will set this variable to zero. Second, if you have a main module, but the Loader still sets this variable and the stack$offset to zero anyway, then the loaded code dynamically initializes the SP and SS registers.

stack$size — The Loader sets this WORD to the number of bytes required for the loaded program's stack. The calling task can initialize the stack pointer (SP register) to the sum of stack$offset and stack$size. The calling task can do this by invoking the CREATE$TASK or the CREATE$JOB system call.

DESCRIPTION
stack$size (continued)

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

init$ds — This is the value that your task should use to initialize the DS (data segment) register. The Loader will set this value to zero if the object file contains no main module. If the object file contains a main mudule and the Loader still sets this value to zero, then the loaded code dynamically sets the DS register.

CONDITION CODES

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

Sequential Condition Codes

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

E$OK — No exceptional conditions.

EBADHEADER — The object file being loaded does not begin with a header record for a loadable object module.

E$CHECKSUM — The header record of the object file contains a checksum error.

E$CONTEXT — This exception code can be caused by any of the following circumstances:

- The calling task specified a connection that was already open.

CONDITION CODES
E$CONTEXT (continued)

- The calling task specified a connection for a device rather than for a named file.
- The Loader opened the connection but some other task closed the connection before the loading operation was begun.

E$EXIST — The calling task specified a connection that has been deleted or is in the process of being deleted.

E$FACCESS — The calling task specified a connection that does not have READ access to the object file.

E$FLUSHING — The device containing the file to be loaded is being detached.

E$IO — An I/O error occurred.

E$LIMIT — This exception code can be caused by any of the following circumstances:

- The job containing the calling task has reached its object limit.
- Either the calling task's job, or the job's default user object, is currently involved in more than 255 (decimal) I/O operations.

E$LOADER$SUPPORT — The object file requires capabilities not configured into the Application Loader. For example, you might be attempting to load PIC with a Loader configured only for absolute code.

E$MEM — This exception code can be caused by any of the following circumstances:

- The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
- The memory pool of the Basic I/O System's job does not currently have a block of memory large enough to allow this system call to run to completion.

ENOTCONFIGURED — One or more of the following system calls was not incorporated into the system during the configuration process:

CONDITION CODES
ENOTCONFIGURED (continued)

A$CLOSE (Basic I/O System)
A$LOAD (Application Loader)
A$OPEN (Basic I/O System)
A$READ (Basic I/O System)
CREATE$MAILBOX (Nucleus)
CREATE$SEGMENT (Nucleus)
CREATE$TASK (Nucleus)
GET$TYPE (Nucleus)
RECEIVE$MESSAGE (Nucleus)

E$SHARE	The calling task specified a connection for a file that is already being used by some other task, and the Application Loader is unable to share the file.
E$SUPPORT	The calling task specified a connection that was not created in the calling task's job.
E$TYPE	The connection parameter refers to an object that is not a connection.

Concurrent Condition Codes

Once the Application Loader has actually begun the loading operation, it returns condition codes through the except$code field of the Loader Result Segment. The Loader can return the following condition codes in this manner.

EBADGROUP	The object file being loaded contains an invalid group definition record.
EBADSEGMENT	The object file being loaded contains an invalid segment definition record.
E$CHECKSUM	At least one record of the file being loaded contains a checksum error.
E$EOF	The Application Loader encountered an unexpected end of file.
E$EXIST	This exception code can be caused by any of the following circumstances:

- The mailbox specified in the response$mbox parameter was deleted before the loading operation was completed.

CONDITION CODES
E$EXIST (continued)

- The device containing the file to be loaded was detached before the loading operation was completed.

E$FIXUP	The object file contains an invalid fixup record.
E$FLUSHING	The device containing the file to be loaded is being detached.
E$IO	An I/O error occurred during the loading operation.
E$LIMIT	The job containing the calling task has reached its object limit.
ENOLOADER$MEM	This exception code can be caused by any of the following circumstances: • The memory pool of the calling task's job does not currently have a block of memory large enough to allow the Application Loader to run. • The memory pool of the Basic I/O System's job does not currently have a block of memory large enough to allow the Application Loader to run.
ENOMEM	The Application Loader is attempting to load PIC or LTL groups or segments, but the memory pool of the calling task's job does not currently contain a block of memory large enough to accommodate these groups or segments.
E$NOSTART	The object file does not specify the entry point for the program being loaded.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: A$ATTACH$FILE (Basic I/O System) A$SEEK (Basic I/O System) CREATE$TASK (Nucleus) DELETE$TASK (Nucleus) EXITIOJOB (Extended I/O System)
E$PARAM	The object file being loaded has a stack smaller than 16 bytes.

A$LOAD (continued)

CONDITION CODES (continued)

ERECFORMAT	At least one record in the file being loaded contains a format error.
ERECLENGTH	The file being loaded contains a record that is longer than the Loader's internal buffer. The Loader's buffer length is a parameter specified during the configuration of the Loader.
ERECTYPE	This exception code can be caused by any of the following circumstances: • At least one record in the file being loaded is of a type that the Loader cannot process. • The Loader has encountered records in a sequence that the it cannot process.

When the A$LOAD system call is used, the concurrent part of the system call can fail without returning an exception code. This can happen only when the SET$EXCEPTION$HANDLER system call was not incorporated into the system during the configuration process.

The A$LOAD$IO$JOB system call creates an I/O job and creates a task within the job. This system call asynchronously loads the code for the task from secondary storage.

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

INPUT PARAMETERS

connection — is a WORD containing a token for a connection to the file that the Application Loader will load. The connection must satisfy four requirements:

- It must be a connection for a named file.
- It must be closed.
- It must have READ access.
- It must have been created in the calling task's job.

pool$lower$bound — is a WORD containing a value that the Loader uses to compute the pool size for the new I/O job. See the following description for details.

pool$upper$bound — is a WORD containing a value that the Loader uses to compute the pool size for the new I/O job. See the following description for details.

except$handler — is a POINTER to a structure that specifies the new job's exception handler. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

job$flags — is a WORD that tells the Nucleus whether to check the validity of objects used as parameters in system calls. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

task$priority — is a BYTE that specifies the priority of the loaded task in the new job. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

INPUT PARAMETERS (continued)

task$flags — is a WORD that tells the Nucleus of any special capabilities that should be accorded tasks in the new I/O job. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

msg$mbox — is a WORD containing a token for a mailbox that serves two purposes. The first purpose is to receive the Loader Result Segment after the loading operation is completed. The format of the Loader Result Segment is provided later in this description.

The second purpose is to receive a message from the newly created I/O job. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about the second purpose.

OUTPUT PARAMETERS

except$ptr — is a POINTER to a WORD in which the Application Loader will place the condition code generated by the sequential part of the system call.

job — is a WORD in which the Application Loader will place the token for the newly created I/O job. This token is valid only if the Application Loader returns an E$OK condition code to the WORD specified by the except$ptr parameter.

DESCRIPTION

This system call operates in two phases. The first phase occurs during the sequential part of this system call. (Refer to Chapter 6 for a discussion of the sequential and concurrent parts of an asynchronous system call.) During this first phase, the Application Loader accomplishes three things:

1. It creates an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a definition of I/O jobs.

2. It validates the header record of the object file.

3. It returns a condition code that reflects the success or failure of the first phase. The Application Loader places this condition code in the WORD to which the except$ptr parameter points.

DESCRIPTION (continued)

The second phase occurs during the concurrent part of the system call. It accomplishes three things:

1. It loads the file designated by the connection parameter.

2. It places the loaded program into execution as a task in the new job.

3. It sends a Loader Result Segment to the mailbox specified by the msg$mbox parameter. This segment contains, among other things, a condition code that indicates the success or failure of the second phase.

Pool Size for the New Job

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

- pool$lower$bound parameter.

- pool$upper$bound parameter.

- an Application Loader configuration parameter that specifies the default dynamic memory requirements. The name of this parameter is L$DEFAULT$MEMPOOL, and it is explained in detail in the chapter of the iRMX 86 CONFIGURATION GUIDE that describes the process of configuring the Application Loader.

- memory requirements specified in the object file.

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

1. You can set both pool$lower$bound and pool$upper$bound to zero. If you do this, the Loader will decide how large a pool to allocate to the new I/O job. The Loader uses the requriements of the object file and L$DEFAULT$MEMPOOL to make this decision.

2. You can set pool$upper$bound to FFFFh. If you do this, the Loader will allow the new I/O job to borrow memory from the calling task's job, and the size of the memory pool will be as in Option 1.

3. You can use either or both of the bound parameters to override the Loader's decision on pool size. If the Loader's decision lies outside the bound(s) that you specify, the Loader will readjust it to comply with your bounds.

DESCRIPTION (continued)

Be aware that if you select options 1 or 3, the Loader will create an I/O job with min$pool$size equal to max$pool$size. This means that the new I/O job will not be able to borrow memory from the calling task's job. If you want the I/O job to be able to borrow memory, select Option 2.

Asynchronous Behavior

This system call is asynchronous. It allows the calling task to continue running while the loading operation is in progress. When the loading operation is finished, the Application Loader will send a Loader Result Segment to the mailbox designated by the msg$mbox parameter. Refer to Chapter 6 for a detailed description of asynchronous behavior.

File Sharing

The Application Loader does not expect exclusive access to the file. However, if another task is also using the file, that task must access the file only for reading.

Format of the Loader Result Segment

The Loader Result Segment has the following form:

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

where:

termination$code is a WORD in which the Application Loader places one of two values. A value of 100h indicates that the loading operation was successful. A value of 2 indicates that the loading operation was a failure.

In case of failure, you must delete the newly created I/O job because the Loader does not.

DESCRIPTION (continued)

except$code — is a WORD in which the Application Loader will place the concurrent condition code. Possible values and interpretations are provided later in this description.

job$token — is a WORD in which the Application Loader will place the token for the newly created I/O job.

return$data$len — is a BYTE that is always set to 9.

record$count — is a WORD containing the number of object records read by the Application Loader. If the loading operation terminates, this value will indicate the last record read.

errorrectype — is a BYTE that identifies the type of record causing termination of the loading operation. A value of zero means that no record caused termination.

num$undefined$refs — An external fixup usually (but not always) indicates an error during the linking process. The Loader will continue to run even if an object file contains external fixups. The purpose of this num$undefined$refs depends upon the kind of code that your Application Loader is configured to load:

- If the Loader is configured for LTL code, this WORD contains the number of external fixups that the loader detected during the loading operation.
- If the Loader is configured to load PIC or absolute code, the Loader will set this WORD to 1 or 0. If the Loader found no external fixups during the loading operation, this WORD will be set to 0. If external fixups were found, this WORD will be set to 1.

mem$requested — is a WORD whose value indicates the number of 16-byte paragraphs that the object file requested for the new job. This request included the memory needed for all segments and for the job's memory pool.

mem$received — is a WORD whose value indicates the number of 16-byte pages actually allocated to the new job.

DESCRIPTION (continued)

Restriction

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

CONDITION CODES

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
EBADHEADER	The object file being loaded does not begin with a header record for a loadable object module.
E$CHECKSUM	The header record of the object file contains a checksum error.
E$CONTEXT	This exception code can be caused by any of the following circumstances: • The calling task specified a connection that was already open. • The calling task specified a connection for a device rather than for a named file. • The Loader opened the connection but some other task closed the connection before the loading operation was begun. • The calling task's job is not an I/O job.

CONDITION CODES (continued)

E$EXIST	This exception code can be caused by any of the following circumstances: • The calling task specified a connection that has been deleted or is in the process of being deleted. • The calling task's job has no global job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a definition of global job. • The msg$mbox parameter does not refer to an existing object.
E$FACCESS	The connection supplied by the calling task does not have READ access to the object file.
E$FLUSHING	The device containing the object file is being detached.
E$IO	An I/O error occurred.
EJOBPARAM	The pool$upper$bound parameter is both nonzero and smaller than the pool$lower$bound parameter.
EJOBSIZE	The pool$upper$bound parameter is nonzero and too small for the object file to be loaded.
E$LIMIT	This exception code can be caused by any of the following circumstances: • The object limit of the Basic I/O System's job has been reached. This limit is set during the configuration process. • Either the newly created I/O job or its default user object is involved in more than 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$LOADER$SUPPORT	The object file requires capabilities not configured into the Application Loader. For example, you might be attempting to load PIC code with a loader configured only for absolute code.

A$LOAD$IO$JOB (continued)

CONDITION CODES (continued)

E$MEM — This exception code can be caused by any of the following circumstances:

- The memory pool of the calling task's job does not currently have a block of memory large enough to allow the creation of the new I/O job.
- The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the initial task to start running.
- The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow the object file to be loaded.

ENOLOADER$MEM — The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the loader to run.

ENOTCONFIGURED — At least one of the following system calls was not incorporated into the system during the configuration process:

A$CLOSE (Basic I/O System)
A$LOAD$IO$JOB (Application Loader)
A$OPEN (Basic I/O System)
A$READ (Basic I/O System)
CATALOG$OBJECT (Nucleus)
CREATEIOJOB (Extended I/O Job)
CREATE$MAILBOX (Nucleus)
CREATE$SEGMENT (Nucleus)
GET$TYPE (Nucleus)
RECEIVE$MESSAGE (Nucleus)

E$PARAM — The value of the except$mode field within the except$handler structure lies outside the range of 0 - 3, inclusive.

E$SHARE — The calling task specified a connection for a file that is already being used by some other task, and the Application Loader is unable to share the file.

E$SUPPORT — The calling task specified a connection that was not created in the calling task's job.

CONDITION CODES (continued)

E$TIME	The calling task's job is not an I/O job.
E$TYPE	The connection parameter does not refer to a connection object.

Concurrent Condition Codes

Once the Application Loader has actually begun the loading operation, it returns condition codes through the except$code field of the Loader Result Segment. The Loader can return the following condition codes in this manner.

EBADGROUP — The object file being loaded contains an invalid group definition record.

EBADSEGMENT — The object file being loaded contains an invalid segment definition record.

E$CHECKSUM — At least one record of the file being loaded contains a checksum error.

E$EOF — The Application Loader encountered an unexpected end of file.

E$EXIST — This exception code can be caused by any of the following circumstances:

- The mailbox specified in the msg$mbox parameter was deleted before the loading operation was completed.
- The device containing the file to be loaded was detached before the loading operation was completed.

E$FACCESS — The default user of the newly created I/O job does not have READ access to the object file.

E$FIXUP — The object file contains an invalid fixup record.

E$FLUSHING — The device containing the file to be loaded is being detached.

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

A$LOAD$IO$JOB (continued)

CONDITION CODES (continued)

E$LIMIT — This exception code can be caused by any of the following circumstances:

- The job containing the calling task is not an I/O job.
- The Basic I/O System's job has reached its object limit. This object limit was specified during the configuration of the Basic I/O System. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.
- The value of the task$priority parameter is greater than the newly created I/O job's maximum priority. This maximum priority was specified during the configuration of the Extended I/O System. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.
- The object directory of the newly created I/O job is full. The size of this object directory was specified during the configuration of the Extended I/O System. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.
- Either the newly created I/O job, or its default user, is currently involved in more than 255 (decimal) I/O operations.

ENOLOADER$MEM — This exception code can be caused by any of the following circumstances:

- The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Application Loader to run.
- The memory pool of the Basic I/O System's job does not currently have a block of memory large enough to allow the Application Loader to run.

ENOMEM — The Application Loader is attempting to load PIC or LTL groups or segments, but the memory pool of the newly created I/O job does not currently contain a block of memory large enough to accommodate these groups or segments.

CONDITION CODES (continued)

E$NOSTART — The object file does not specify the entry point for the program being loaded.

ENOTCONFIGURED — At least one of the following system calls was not incorporated into the system during the configuration process:

A$ATTACH$FILE (Basic I/O System)
A$SEEK (Basic I/O System)
CATALOG$OBJECT (Nucleus)
CREATE$TASK (Nucleus)
DELETE$TASK (Nucleus)
EXITIOJOB (Extended I/O System)
UNCATALOG$OBJECT (Nucleus)

E$PARAM — The object file being loaded has a stack smaller than 16 bytes.

ERECFORMAT — At least one record in the file being loaded contains a format error.

ERECLENGTH — The file being loaded contains a record that is longer than the Loader's maximum record length. The Loader's maximum record length is a parameter specified during the configuration of the Loader. Refer to the iRMX 86 CONFIGURATION GUIDE for details.

ERECTYPE — This exception code can be caused by any of the following circumstances:

- At least one record in the file being loaded is of a type that the Loader cannot process.
- The Loader has encountered records in a sequence that it cannot process.

When the A$LOAD$IO$JOB system call is used, the concurrent part of the system call can fail without returning an exception code. This can happen only when the SET$EXCEPTION$HANDLER system call was not incorporated into the system during the configuration process.

S$LOAD$IO$JOB

The S$LOAD$IO$JOB system call creates an I/O job containing one task. The code for the task is a program loaded from secondary storage.

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

INPUT PARAMETERS

path$ptr	is a POINTER to a STRING containing a path name for the named file that contains the object code to be loaded. This path name must conform to the Extended I/O System path syntax for named files. If you are unfamiliar with the path syntax, refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.
pool$lower$bound	is a WORD containing a value that the Loader uses to compute the pool size for the new I/O job. See the following description for details.
pool$upper$bound	is a WORD containing a value that the Loader uses to compute the pool size for the new I/O job. See the following description for details.
except$handler	is a POINTER to a structure that specifies the new job's exception handler. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.
job$flags	is a WORD that tells the Nucleus whether to check the validity of objects used as parameters in system calls. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.
task$priority	is a BYTE that specifies the priority of the loaded task in the new job. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.
task$flags	is a WORD that tells the Nucleus of any special capabilities that should be accorded tasks in the new I/O job. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information.

INPUT PARAMETERS (continued)

msg$mbox — is a WORD containing a token for a mailbox that is to receive a termination message from the newly created I/O job. Refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about the second purpose.

OUTPUT PARAMETERS

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

job — is a WORD in which the Application Loader will place the token for the newly created I/O job. This token is valid only if the Application Loader returns an E$OK condition code to the WORD specified by the except$ptr parameter.

DESCRIPTION

This system call creates an I/O job, loads the specified file, and places the loaded code into execution as a task within the new I/O job.

Synchronous Behavior

This system call is synchronous. The calling task resumes running only after the system call has succeeded or failed in its attempt to create a task that uses the code from the specified file.

File Sharing

The Application Loader does not expect exclusive access to the file. However, if another task is also using the file, that task must access the file only for reading.

Pool Size for the New Job

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

S$LOAD$IO$JOB (continued)

DESCRIPTION (continued)

- pool$lower$bound parameter.
- pool$upper$bound parameter.
- an Application Loader configuration parameter that specifies the default dynamic memory requirements. The name of this parameter is L$DEFAULT$MEMPOOL, and it is explained in detail in the chapter of the iRMX 86 CONFIGURATION GUIDE that describes the process of configuring the Application Loader.
- memory requirements specified in the object file.

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

1. You can set both pool$lower$bound and pool$upper$bound to zero. If you do this, the Loader will decide how large a pool to allocate to the new I/O job. The Loader uses the requriements of the object file and L$DEFAULT$MEMPOOL to make this decision.

2. You can set pool$upper$bound to FFFFh. If you do this, the Loader will allow the new I/O job to borrow memory from the calling task's job, and the size of the memory pool will be as in Option 1.

3. You can use either or both of the bound parameters to override the Loader's decision on pool size. If the Loader's decision lies outside the bound(s) that you specify, the Loader will readjust it to comply with your bounds.

Be aware that if you select options 1 or 3, the Loader will create an I/O job with min$pool$size equal to max$pool$size. This means that the new I/O job will not be able to borrow memory from the calling task's job. If you want the I/O job to be able to borrow memory, select Option 2.

Restriction

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

CONDITION CODES

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

CONDITION CODES (continued)

E$OK	No exceptional conditions.
EBADGROUP	The object file being loaded contains an invalid group definition record.
EBADHEADER	The object file being loaded does not begin with a header record for a loadable object module.
EBADSEGMENT	The object file being loaded contains an invalid segment definition record.
E$CHECKSUM	At least one object record in the file being loaded contains a checksum error.
E$CONTEXT	The calling task's job is not an I/O job.
E$EOF	The Application Loader encountered an unexpected end of file.
E$EXIST	This exception code can be caused by any of the following circumstances:

- The mailbox specified by the msg$mbox parameter was deleted while the system call was running.
- The calling task's job has no global job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for a definition of global job.
- The msg$mbox parameter does not refer to an existing object.
- The device containing the object file was detached.

E$FACCESS	The default user object for the new I/O job does not have READ access to the specified file. This user object is specified during the process of configuring the Extended I/O System. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.
E$FIXUP	The object file contains an invalid fixup record.
E$FNEXIST	This exception code can be caused by any of the following circumstances:

S$LOAD$IO$JOB (continued)

CONDITION CODES
E$FNEXIST (continued)

- Either the specified object file, or some file in the specified path, does not exist.
- Either the specified object file, or some file in the specified path, is marked for deletion.

E$FLUSHING — The device containing the object file is being detached.

E$IO — An I/O error occurred.

EJOBPARAM — The pool$upper$bound parameter is both nonzero and smaller than the pool$lower$bound parameter.

EJOBSIZE — The pool$upper$bound parameter is nonzero and too small for the object file to be loaded.

E$LIMIT — This exception code can be caused by any of the following circumstances:

- The calling task's job is not an I/O job.
- The object limit of the Basic I/O System's job has been reached. This limit is set during the process of configuring the Basic I/O System. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.
- The value of the task$priority parameter is greater than the new I/O job's maximum priority. This maximum priority is set during the process of configuring the Extended I/O System. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.
- The object directory of the new I/O job is full. The size of this object directory is set during the process of configuring the Extended I/O System. Refer to the iRMX 86 CONFIGURATION GUIDE for more information.
- Either the newly created I/O job or its default user object is involved in more than 255 (decimal) I/O operations.
- The calling task's job is not an I/O job.

CONDITION CODES (continued)

E$LOADER$SUPPORT	The object file requires capabilities that are not configured into the Application Loader. For example, you might be attempting to load PIC with a loader configured only for absolute code.
E$MEM	This exception code can be caused by any of the following circumstances: • The memory pool of the calling task's job does not currently have a block of memory large enough to allow the creation of the new I/O job. • The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the initial task to start running. • The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow the object file to be loaded.
ENOLOADER$MEM	This exception code can be caused by any of the following circumstances: • The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Loader to run. • The memory pool of the Basic I/O Job does not currently have a block of memory large enough to allow the Loader to run.
E$NOMEM	The object file contains either PIC segments or groups, or LTL segments or groups. In any case, the memory pool of the new I/O job does not have a block of memory large enough to allow the Application Loader to load these records.
E$NOSTART	The object file does not specify the entry point for the program being loaded.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process:

S$LOAD$IO$JOB (continued)

CONDITION CODES
ENOTCONFIGURED (continued)

A$ATTACH$FILE (Basic I/O System)
A$CLOSE (Basic I/O System)
A$OPEN (Basic I/O System)
A$READ (Basic I/O System)
A$SEEK (Basic I/O System)
CATALOG$OBJECT (Nucleus)
CREATEIOJOB (Extended I/O Job)
CREATE$MAILBOX (Nucleus)
CREATE$SEGMENT (Nucleus)
CREATE$TASK (Nucleus)
DELETE$TASK (Nucleus)
EXITIOJOB (Extended I/O System)
GET$TYPE (Nucleus)
RECEIVE$MESSAGE (Nucleus)
S$ATTACH$FILE (Extended I/O System)
S$LOAD$IO$JOB (Application Loader)
UNCATALOG$OBJECT (Nucleus)

E$PARAM — This exception code can be caused by any of the following circumstances:

- The value of the except$mode field within the except$handler structure lies outside the range of 0 - 3, inclusive.
- The object file being loaded requested a stack smaller than 16 bytes.

ERECFORMAT — At least one record in the file being loaded contains a format error.

ERECLENGTH — The file being loaded contains a record that is longer than the Loader's maximum record length. The Loader's maximum record length is a parameter specified during the configuration of the Loader. Refer to the iRMX 86 CONFIGURATION GUIDE for details.

ERECTYPE — This exception code can be caused by any of the following circumstances:

- At least one record in the file being loaded is of a type that the Loader cannot process.
- The Loader has encountered records in a sequence that the Loader cannot process.

E$TIME — The calling task's job is not an I/O job.

The S$OVERLAY system call is invoked by a root module to load an overlay module.

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

INPUT PARAMETER

name$ptr	is a POINTER to a STRING that contains the name of an overlay. Refer to Chapter 5 for an explanation of overlays.

OUTPUT PARAMETER

except$ptr	is a POINTER to a WORD in which the Application Loader will place an exception code.

DESCRIPTION

This system call is invoked by a root module whenever the root module wishes to load an overlay module.

Synchronous Behavior

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

File Sharing

The Application Loader does not expect exclusive access to the file containing the overlay module. However, if another task is also using the file, the task must access the file only for reading.

CONDITION CODES

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

CONDITION CODES (continued)

E$OK	No exceptional conditions.
E$CHECKSUM	At least one object record in the overlay being loaded contains a checksum error.
E$EOF	The Application Loader encountered an unexpected end of file.
E$EXIST	The device containing the object file was detached during the loading operation.
E$FIXUP	The object file contains an invalid fixup record.
E$FLUSHING	The device containing the object file is being detached.
E$IO	An I/O error occurred during the loading operation.
E$LIMIT	This exception code can be caused by any of the following circumstances: • The calling task's job is not an I/O job. • Either the calling task's job, or its default user object, is currently involved in more than 255 (decimal) I/O operations.
E$NOMEM	The overlay module contains either PIC segments or groups, or LTL segments or groups. In any case, the memory pool of the new I/O job does not have a block of memory large enough to allow the Application Loader to load the overlay module.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the process of configuration: LOOKUP$OBJECT (Nucleus) S$OVERLAY (Application Loader)
ERECFORMAT	At least one record in the overlay being loaded contains a format error.
ERECLENGTH	The overlay being loaded contains a record that is longer than the Loader's maximum record length. The Loader's maximum record length is a parameter specified during the configuration of the Loader. Refer to the iRMX 86 CONFIGURATION GUIDE for details.

CONDITION CODES (continued)

ERECTYPE — This exception code can be caused by any of the following circumstances:

- At least one record in the overlay being loaded is of a type that the Loader cannot process.
- The Loader has encountered records in a sequence that it cannot process.

E$OVERLAY — The overlay name indicated by the name$ptr parameter is not defined in the root module.

APPENDIX A. DATA TYPES

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

BYTE	An unsigned, eight-bit binary number.
WORD	An unsigned, two-byte binary number.
INTEGER	A signed, two-byte, binary number. Negative numbers are stored in two's-complement form.
OFFSET	A word whose value represents the distance (in bytes) from the base of a segment.
TOKEN	A word whose value identifies an object.
POINTER	Two consecutive words containing the base of a segment and an offset into the segment. The offset must be in the word having the lower address.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes that follow it in the string. Each of the bytes except for the first contains an ASCII-encoded character.

APPENDIX B. CONDITION CODES

The iRMX 86 Application Loader uses two kinds of condition codes to inform your tasks of any problems that occur during the execution of a system call -- sequential condition codes and concurrent condition codes. The distinguishing feature between the two kinds of codes is the method that the Application Loader uses to return the code to the calling task. For a discussion of the difference between these kinds of codes, refer to Chapter 7.

The meaning of a specific condition code depends upon the system call that returns the code. For this reason, this appendix does not list interpretations. Refer to Chapter 7 if you want to interpret the codes.

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

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

Be aware that this list is not complete. Any exception codes not included in this list can be found in the apendixes of one of the following manuals:

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

NORMAL CONDITION CODE

NAME OF CONDITION	HEXADECIMAL VALUE
E$OK	0h

PROGRAMMING EXCEPTION CODES

NAME OF CONDITION	HEXADECIMAL VALUE
EJOBPARAM	8060h

ENVIRONMENTAL EXCEPTION CODES

EABSADDRESS	60h
EBADGROUP	61h
EBADHEADER	62h
EBADSEGDEF	63h
E$CHECKSUM	64h
E$EOF	65h
E$FIXUP	66h
EJOBSIZE	6Dh
E$LOADER$SUPPORT	6Fh
ENOLOADER$MEM	67h
ENOMEM	68h
ENOSTART	6Ch
ENOTCONFIGURED	8h
E$OVERLAY	6Eh
ERECFORMAT	69h
ERECLENGTH	6Ah
ERECTYPE	6Bh

INDEX

Underscored entries are primary references.

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

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

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

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. ____________

NAME ______________________________ DATE ____________

TITLE ______________________________

COMPANY NAME/DEPARTMENT ______________________________

ADDRESS ______________________________

CITY ____________________ STATE ________ ZIP CODE ____________

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Pkwy.
Hillsboro, Oregon 97123

O.M.S. Technical Publications

intel®

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

Printed in U.S.A.