iRMX™ 86 APPLICATION LOADER REFERENCE MANUAL

CONTENTS

CONTENTS (continued)

***

CHAPTER 1
INTRODUCTION TO THE APPLICATION LOADER

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

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

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

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

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

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

LOADER TERMINOLOGY

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

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

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

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

OBJECT CODE

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

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

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

TYPES OF OBJECT CODE

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

Absolute Code

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

Position-Independent Code (PIC)

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

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

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

Load-Time-Locatable (LTL) Code

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

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

SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

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

An 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, refer to Appendix C.

I/O JOB

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

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

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

OVERLAY

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

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

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

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

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

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

LOADER FEATURES

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

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

The following sections briefly discuss each of these features.

DEVICE INDEPENDENCE

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

SYNCHRONOUS AND ASYNCHRONOUS SYSTEM CALLS

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

SUPPORT FOR OVERLAID PROGRAMS

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

CONFIGURABILITY

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

PREPARING CODE FOR LOADING

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

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

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

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

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

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

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

INVOKING iRMX™ 86 SYSTEM CALLS

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

ENTRY POINTS

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

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

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

Using A Main Module

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

Writing A Procedure To Be Loaded By The Loader

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

STACK SIZES

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

HOW THE LOADER WORKS

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

***

CHAPTER 2
APPLICATION LOADER SYSTEM CALLS

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

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

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

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

RESPONSE MAILBOX PARAMETER

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

In general the Loader Result Segment indicates the result of the loading operation. The format of a Loader Result Segment depends upon which system call was invoked, so details about Loader Result Segments are included in descriptions of the A$LOAD and A$LOAD$IO$JOB system calls.

Avoid using the same response mailbox for more than one concurrent invocation of asynchronous system calls. This is necessary because it is possible for the Loader to return Loader Result Segments in an order different than the order of invocation. On the other hand, it is safe to use the same mailbox for multiple invocations of asynchronous system calls if only one task invokes the calls and the task always obtains the result of one call via RQ$RECEIVE$MESSAGE before making the next call.

CONDITION CODES

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

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

CONDITION CODES FOR SYNCHRONOUS SYSTEM CALLS

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

CONDITION CODES FOR ASYNCHRONOUS SYSTEM CALLS

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

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

Sequential Condition Codes

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

Concurrent Condition Codes

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

SYSTEM CALL DICTIONARY

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

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

A$LOAD

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

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

INPUT PARAMETERS

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

OUTPUT PARAMETER

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

DESCRIPTION

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

Using A$LOAD to Load a Main Module

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

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

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

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

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

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

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

Using A$LOAD To Load A Procedure

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

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

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

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

Asynchronous Behavior

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

File Sharing

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

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

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

Considerations Relating To Code Type

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

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

Effects Of Model Of Segmentation

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

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

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

Format Of The A$LOAD Loader Result Segment

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

The Loader Result Segment has the following form:

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

where:

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

	The value of undefined$ref depends upon your configuration of the Loader. (See Chapter 3 and the iRMX 86 CONFIGURATION GUIDE for information about configuring the Loader.) • If the Loader is configured to load LTL and overlay code, as well as PIC and absolute code, undefined$ref contains the number of undefined external references detected during the loading operation. (Note that undefined$ref equals the number of undefined external references even if the Loader is loading PIC or absolute code.) • If the Loader is configured to load only absolute code or only PIC or absolute code, the Loader sets undefined$ref to 1 or to 0. It is 1 if the Loader finds undefined external references; otherwise, it is 0.
init$ip	A WORD containing the initial value for the loaded program's instruction pointer (IP register). The calling task can use this information in either of two ways: • When invoking the CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system call. • As the destination of a jump within the code segment of the loaded program. Init$ip is 0 if the file does not specify an initial value for the instruction pointer, as can happen when the file contains no main module.
code$seg$base	A WORD or SELECTOR containing the base address for the code segment with the entry point. The value in code$seg$base can be used with init$ip as a POINTER to the entry point of the loaded program. The Loader places 0 into this field if the loaded program does not contain a main module. If you are using a compiler that supports the data type SELECTOR, code$seg$base should be declared a SELECTOR.
stack$offset	A WORD containing the offset of the bottom of the stack, relative to the beginning of the stack segment. The calling task can use the sum of this value and the stack$size to initialize the stack pointer (SP register). The Loader sets stack$offset to zero under each of these circumstances:

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

stack$seg$base	A WORD or SELECTOR containing the base of the stack segment for the loaded program. The calling task can use this value to initialize the stack segment (SP register). Stack$seg$base should be declared a SELECTOR if your compiler supports the SELECTOR data type. The Loader sets stack$seg$base to 0 under each of these circumstances: • If there is no main module. (In this case, the target file does not specify a stack base). • If the loaded code is a main module that dynamically initializes the SP and SS registers.
stack$size	A WORD specifying the number of bytes required for the loaded program's stack. The calling task can initialize the stack pointer (SP register) to the sum of stack$offset and stack$size when invoking the CREATE$TASK, CREATE$JOB, or CREATE$IO$JOB system call. The Loader sets this value to 0 whenever both the stack$offset and stack$seg$base are 0. When all three stack-related parameters are 0 and the target file contains a main module, the loaded code must set the stack pointer (SP register) and stack segment (SS register).
data$seg$base	A WORD or SELECTOR containing the initial base address of the data segment (DS register). If your compiler supports the SELECTOR data type, data$seg$base should be declared a SELECTOR. The Loader sets this value to 0 under each of these circumstances: • If the target file contains no main module. • If the main module dynamically sets the DS register after the program starts running.

CONDITION CODES

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

Sequential Condition Codes

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

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

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

Concurrent Condition Codes

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

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

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

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

A$LOAD$IO$JOB

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

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

INPUT PARAMETERS

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

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

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

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

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

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

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

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

task$priority — A BYTE which,

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

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

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

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

Set bits 2 through 15 to 0.

msg$mbox

A TOKEN for a mailbox that serves two purposes. The first purpose is to receive the Loader Result Segment after the loading operation is completed. The format of the Loader Result Segment is provided later in this description.

The second purpose is to receive an exit message from the newly created I/O job. The description of the CREATE$IO$JOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL shows the format of an exit message.

OUTPUT PARAMETERS

except$ptr

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

job

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

DESCRIPTION

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

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

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

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

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

Restriction

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

Pool Size For The New Job

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

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

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

1. You can set both pool$lower$bound and pool$upper$bound to 0. If you do this, the Loader decides how large a pool to allocate to the new I/O job. The Loader uses the requirements of the target file and the default memory pool size -- established when the system is configured -- to make this decision. Unless you have unusual requirements, you should choose this option.

2. You can use either or both of the bound parameters to override the Loader's decision on pool size. If the Loader's decision lies outside the bound(s) that you specify, the Loader adjusts its decision so that it complies with your bounds.

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

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

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

Format Of The Loader Result Segment

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

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

where:

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

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

CONDITION CODES

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

Sequential Condition Codes

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

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

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

Concurrent Condition Codes

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

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

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

S$LOAD$IO$JOB

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

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

INPUT PARAMETERS

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

STRUCTURE

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

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

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

except$handler (continued)

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

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

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

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

task$priority — A BYTE which,

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

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

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

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

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

Set bits 2 through 15 to 0.

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

OUTPUT PARAMETERS

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

DESCRIPTION

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

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

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

Pool Size For The New Job

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

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

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

1. You can set both pool$lower$bound and pool$upper$bound to zero. If you do this, the Loader decides how large a pool to allocate to the new I/O job. The Loader uses the requirements of the target file and the default memory pool size -- established when the system is configured -- to make this decision. Unless you have unusual requirements, you should choose this option.

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

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

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

CONDITION CODES

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

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

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

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

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

S$OVERLAY

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

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

INPUT PARAMETER

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

OUTPUT PARAMETER

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

DESCRIPTION

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

Synchronous Behavior

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

File Sharing

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

CONDITION CODES

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

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

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

***

CHAPTER 3 CONFIGURATION OF THE APPLICATION LOADER

The Application Loader is a configurable layer of the Operating System. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, the iRMX 86 manual set provides three kinds of information:

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

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

TYPES OF JOB-LOADING SYSTEM CALLS

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

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

LOADER IN ROM

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

TYPE OF CODE TO BE LOADED

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

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

DEFAULT MEMORY POOL SIZE

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

SIZE OF APPLICATION LOADER BUFFERS

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

***

APPENDIX A DATA TYPES

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

BYTE	An unsigned, eight-bit binary number.
WORD	An unsigned, two-byte, binary number.
INTEGER	A signed, two-byte, binary number. Negative numbers are stored in two's-complement form.
POINTER	Two consecutive words containing the base address of a (64K-byte processor) segment and an offset in the segment. The offset is in the word having the lower address.
OFFSET	A word whose value represents the distance from the base address of a segment.
SELECTOR	The base address of a segment.
TOKEN	A word or selector whose value identifies an object. A token can be declared literally a WORD or a SELECTOR depending on your needs.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes that follow it in the string.
DWORD	A 4-byte unsigned binary number.

***

APPENDIX B
CONDITION CODES

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

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

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

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

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

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

NORMAL CONDITION CODE

NAME OF CONDITION	HEXADECIMAL VALUE
E$OK	OH

PROGRAMMER ERROR CODES

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

ENVIRONMENTAL PROBLEM CODES

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

***

APPENDIX C
ASYNCHRONOUS SYSTEM CALLS

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

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

- the sequential part

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

- the concurrent part

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

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

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

1. Your application issues the A$LOAD system call. (Asynchronous calls require that your application specify a response mailbox for communication with the concurrent part of the system call.)

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

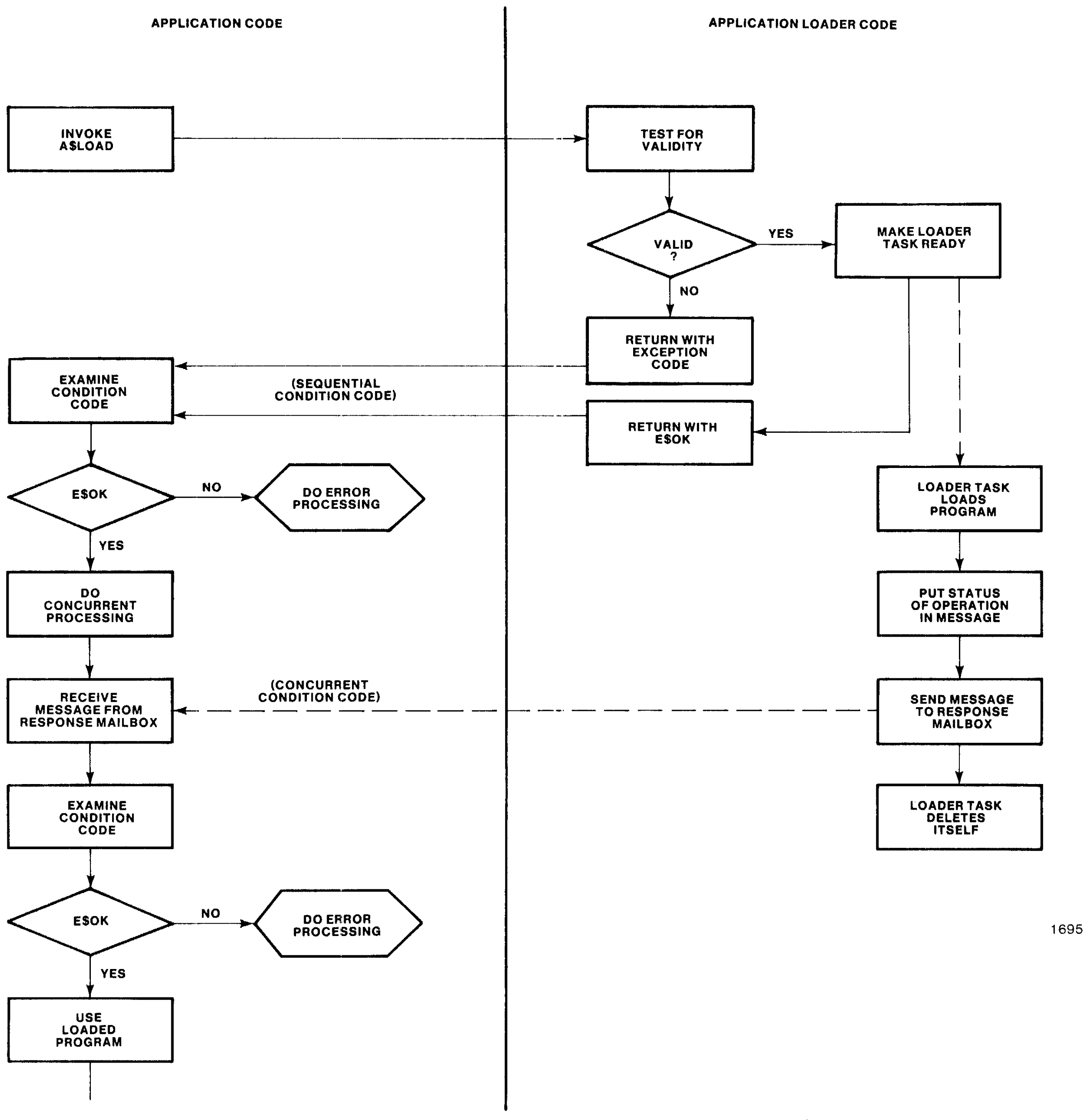

Figure C-1. Behavior Of An Asynchronous System Call

3. If the Operating System detects a problem, it places a sequential exception code in the word to which your except$ptr parameter points. It then returns control to your application. It does not make the Loader task ready.

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

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

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

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

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

In the foregoing example, we used a specific system call (A$LOAD) to show how asynchronous calls allow your application to run concurrently with loading operations. Now let's look at some generalities about all iRMX 86 asynchronous calls:

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

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

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

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

- Whenever the concurrent part of a system call runs, the Loader signals its completion by sending an object to the response mailbox. The precise nature of the object depends upon which system call your application invoked. You can find out what kind of object comes back from a particular system call by looking up the call in Chapter 2 of this manual.

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

***

INDEX

Primary references are underscored.

***