iRMX 86™EXTENDED I/O SYSTEM REFERENCE MANUAL

Manual Number: 143308-001

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

REV.	REVISION HISTORY	PRINT DATE
-001	Original Issue	5/81

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

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

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

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

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

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

BXP
CREDIT
i
ICE
iCS
im
Insite
Intel
Intel
Intelevision
Intellec
iRMX
iSBC
iSBX
Library Manager
MCS
Megachassis
Micromap
Multibus
Multimodule
PROMPT
Promware
RMX/80
System 2000
UPI
μScope

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

PREFACE

This manual documents the Extended I/O System, one of the subsystems available with the iRMX 86™ Operating System. Although it contains some introductory and overview material, it is intended primarily to be a quick reference to system calls, providing detailed descriptions of those system calls avaliable to application programmers. Other system calls, which are reserved for system programmers, are discussed only generally. For more detail regarding the reserved system calls, refer to 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.

Readers need not be familiar with the iRMX 86™ Basic I/O System.

CONVENTIONS

This manual uses a generic shorthand to refer to system calls. For example, S$CREATE$FILE means RQSCREATE$FILE. The actual PL/M-86 external procedure names used to invoke these system calls are shown only in Chapter 8, which lists the detailed calling sequences.

Although Chapter 8 lists only the PL/M-86 calling sequences, you can invoke the system calls from 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. Whenever 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™ System Programmer's Reference Manual	142721
iRMX 86™ Configuration Guide	9803126
iRMX 86™ Installation Guide	9803125
iRMX 86™ Programming Techniques	142982
iRMX 86 Human Interface Reference Manual	9803202
Guide to Writing Device Drivers for the iRMX 86™ Operating System	142926
PL/M-86 Programming Manual for 8080/8085-Based Development Systems	9800466
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
8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems	121627
8086/8087/8088 Macro Assembly Language Reference Manual for 8080/8085-Based Development Systems	121623
8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems	121628
8086/8087/8088 Macro Assembler Operating Instructions for 8080/8085-Based Development Systems	121624

CONTENTS

FIGURES

CHAPTER 1. ORGANIZATION OF THIS MANUAL

This manual is divided into eight chapters and five appendixes. Some of the chapters contain introductory material which you do not need if you are already familiar with the Extended I/O System or if you have used this manual before. One chapter contains reference material to which you will refer as you write your application tasks. The appendixes contain technical information that is of interest to only a few readers.

The manual organization is as follows:

Chapter 1	This chapter describes the organization of the manual. You should read this chapter if you are going through the manual for the first time.
Chapter 2	This chapter describes the differences between the Basic I/O System and the Extended I/O System. You should read this chapter if you are not certain which I/O system best meets your requirements.
Chapter 3	This chapter describes the primary features of the Extended I/O System. You will find this chapter particularly useful if you have not used the Extended I/O System until now.
Chapter 4	This chapter explains some basic terminology associated with the Extended I/O System, including the concepts of system programmer, device, volume, file, and connection. You should read this chapter if you are looking through the manual for the first time or if you are unfamiliar with the Extended I/O System.
Chapter 5 through Chapter 7	These chapters describe named, physical, and stream files and how to use them. You should read one or more of these chapters, depending on the kinds of files your application uses.
Chapter 8	This chapter contains detailed descriptions of the system calls provided by the Extended I/O System. The calls are listed in alphabetical order. When writing your application tasks, you can refer to this chapter for specific information about the format and parameters of each system call.
Appendix A	This appendix defines the formats of the data types used by the Extended I/O System. For example, it explains the format of an iRMX 86 STRING.

Appendix B — This appendix provides a list of the types of objects created by the Extended I/O System. It also discusses the resource requirements of the Extended I/O System.

Appendix C — This appendix contains a list of all the condition codes that the Extended I/O System can return. The codes are listed in alphabetical order, and each entry in the list includes the classification of the code (programmer error or environmental condition) and the numeric value of the code.

Appendix D — The Extended I/O System uses object directories extensively. This appendix tells which entries are used by the Extended I/O System. It also tells you which entries you can change and which entries you can't.

Appendix E — This appendix explains the incompatibilities between the system calls of the Basic I/O System and the system calls of the Extended I/O System.

CHAPTER 2. CHOOSING BETWEEN BASIC I/O AND EXTENDED I/O

The iRMX 86 Operating System provides you with a choice of two I/O systems. One of these, the Extended I/O System, is described in this manual. The other, the Basic I/O System, is described in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

The purpose of this chapter is to explain the reason for having two I/O systems and to briefly describe the differences between them. After reading this chapter, you should be able to decide whether your application requires system calls from both systems or from just one. And if one is sufficient, you should be able to decide which one.

As you read this chapter, bear in mind that the Extended I/O System is built upon the Basic I/O System. In other words, if you choose the Extended I/O System, you must also include the Basic I/O System in your product.

REASON FOR HAVING TWO I/O SYSTEMS

The iRMX 86 Operating System is designed to provide Original Equipment Manufacturers (OEMs) with a variety of features that are useful in building application systems. Many of these features are useful in most, but not all applications. This is especially true of features relating to input and output.

Most applications communicate with external devices such as line printers, terminals, disk drives, bubble memories, or even robots. Even so, not all applications have the same requirements. The iRMX 86 Operating System provides two I/O systems to allow you to choose the one that best satisfies the requirements of your application system. And, in the event that both systems would prove useful in one application, the iRMX 86 Operating System allows you to use them both.

BASIC I/O SYSTEM

The Basic I/O System is the more flexible of the two I/O systems. It provides very powerful capabilities, and it makes few assumptions about the requirements of your application. The following features illustrate the flexibility of the Basic I/O System:

- ALLOWS YOU TO DESIGN YOUR OWN BUFFERING ALGORITHM

  Although many applications require buffered I/O, the Basic I/O System does not automatically provide it. Rather than force one particular approach to buffered I/O on all applications, the Basic I/O System allows you to design and implement your own buffering.

- SUPPLIES ASYNCHRONOUS SYSTEM CALLS

  Rather than making assumptions about whether (and how) you wish to overlap your I/O operations, the Basic I/O System allows you explicitly control the synchronization of the system calls.

- GIVES YOUR TASK CONTROL OF DETAILS

  The system calls of the Basic I/O System involve many parameters. Using these parameters, your tasks can closely tailor the behavior of each system call to match the requirements of your application system.

As these features show, the Basic I/O System emphasizes flexibility rather than ease of use. By preserving flexibility, the Basic I/O System provides I/O features that are useful in a broad range of applications.

Clearly, the Basic I/O System does have drawbacks. Many applications that perform I/O do not need the control of details afforded by the Basic I/O System. For many applications, the amount of time required to develop the application system is more critical than the ability to finely tune its performance. For these applications, the iRMX 86 Operating System provides the Extended I/O System.

EXTENDED I/O SYSTEM

The Extended I/O System is designed to be easier to use than the Basic I/O System. The following features of the Extended I/O System help make it easier to use:

- AUTOMATIC BUFFERING OF I/O OPERATIONS

  The Extended I/O System provides you with automatic buffering of all I/O operations. Aside from specifying how many buffers the Extended I/O System is to use, your tasks need not become involved with buffering. Furthermore, if your application system does not require buffering, your tasks can tell the Extended I/O System to use no buffers.

- SYNCHRONOUS SYSTEM CALLS

  The Extended I/O System provides system calls that are synchronous. By freeing your application software from the burden of explicitly synchronizing system calls, the Extended I/O System reduces the complexity of your application system. This, in turn helps reduce development costs.

Although the system calls of the Extended I/O System are synchronous, your application system can still use overlapped I/O operations. To do so, your tasks need only tell the Extended I/O System to use buffers, and the Extended I/O System will automatically overlap your I/O operations.

- FREES YOUR TASKS OF TEDIOUS DETAILS

  The system calls of the Extended I/O System require fewer parameters than do those of the Basic I/O System. This simplifies your application system and reduces development costs.

MAKING THE DECISION

You are faced with a choice of three alternatives. Should you use the Basic I/O System, the Extended I/O System, or both? In order to make this decision, you must decide if your application system requires the flexiblity and fine tuning of the Basic I/O System, the ease of use of the Extended I/O System, or a combination of both. Before you make the final decision, consider these two factors:

MEMORY REQUIREMENTS

The Basic I/O System software is roughly 12K bytes smaller than the software of the Extended I/O System. So if your application system is pressed for memory and does not require the ease of use provided by the Extended I/O System, consider using the Basic I/O System.

However, if your decide to use the Basic I/O System even though your application system needs the features of the Extended I/O System (such as buffering), you might end up using the entire 12K bytes of memory (along with a lot of valuable engineering time) implementing these very same features on top of the Basic I/O System.

Be aware that using both the Basic I/O and Extended I/O Systems requires no more memory than using the Extended I/O System alone.

PERFORMANCE

Because the Basic I/O System gives your application system control of many details, you can probably tune your application system to run faster with the Basic I/O System than with the Extended I/O System. So if performance is more important than reduced development costs, you should consider using the Basic I/O System.

EXAMPLES

The following examples illustrate the advantages of each of the I/O systems. The analysis in each example is based on the assumption that many copies of the application system are to be produced. This assumption makes memory conservation somewhat more important than it would be if only a few application systems were being built.

APPLICATION SYSTEMS USING LITTLE I/O

Suppose that your application system performs very little I/O. For instance, suppose that its only I/O activity is to occasionally log information to a flexible disk.

Because this application system involves very few I/O-related system calls, the Basic I/O System is preferable to the Extended I/O System. The ease of use provided by the Extended I/O System can save you very little manpower (hence money and time) during development because the I/O-related part of your system requires so few man-hours to develop. This marginal benefit is of less use to your application system than is the 12K bytes of memory saved by using the Basic I/O System.

APPLICATION SYSTEMS USING ONLY SEQUENTIAL I/O

Suppose that your application system requires a substantial amount of sequential I/O. In this type of system, a large amount of your development resources will be expended in support of I/O. This factor should make you consider using the Extended I/O System to reduce your manpower requirements while developing the application system.

A second factor should also steer you toward the Extended I/O System -- the sequential I/O. The buffering scheme used by the Extended I/O System is particularly efficient while performing sequential I/O because it incorporates read-ahead and write-behind algorithms to overlap I/O operations and processing.

These two factors, the amount of manpower required to implement I/O and the sequential nature of the I/O, combine to make the Extended I/O System the best choice for this application system.

HIGH PERFORMANCE APPLICATIONS USING RANDOM I/O

Now suppose that your system performs a large amount of random-access I/O, and suppose that performance is a critical consideration that overrides concerns about conserving memory. You should consider the Extended I/O System because, in this application system, it can substantially reduce your development costs. However, two other factors combine to make the Basic I/O System another reasonable choice.

The first factor is the random nature of the I/O. The read-ahead and write-behind algorithm provided by the Extended I/O System is not particularly efficient in random-access I/O operations.

The second factor is the requirement for performance. Using the Basic I/O System as a foundation, you can build an I/O facility that takes advantage of your application system's knowledge of the organization of data in the files. Although such a facility might be expensive to implement, it should run faster than the Extended I/O System in this application.

So in this case, you must weigh the cost of development against the benefit of better performance. If development costs are more important, you should use the Extended I/O System. If performance is more important, you should use the Basic I/O System. Also, don't ignore the option of using the Extended I/O System to create a prototype application system and then later replacing the Extended I/O System with your custom I/O facility.

SUMMARY

In general, you should consider the Basic I/O System for applications that require very little I/O, or for applications requiring finely tuned performance while doing random-access I/O. In contrast, you should consider the Extended I/O System when development costs are critical, especially in applications that use sequential I/O.

Finally, remember that there are circumstances where you should use both I/O systems. One such situation occurs when your application system uses I/O for several purposes, some of which are best accomplished by the Basic I/O System, and others by the Extended I/O System.

CHAPTER 3. FEATURES OF THE EXTENDED I/O SYSTEM

Because the iRMX 86 Extended I/O System is designed primarily for use by Original Equipment Manufacturers (OEMs), it provides a large number of features -- including some that are not generally found in operating systems aimed at end users. These features include:

- Support for Many Kinds of Devices
- Device Independence
- Three Distinct Kinds of Files
- File Independence
- Separation of File Lookup and File Open Operations
- File Sharing and Access Control
- Buffering with Overlapped I/O
- Logical Names for Files and Devices
- Automatic Detection of Removed Media

The first six of these features are also provided by the Basic I/O System, but the balance of the features are available only with the Extended I/O System.

The purpose of this chapter is to briefly explain each of these features.

SUPPORT FOR MANY KINDS OF DEVICES

The iRMX 86 Extended I/O System supports a wide variety of devices. In order to connect a particular device to the Extended I/O System, you must have a device driver (a collection of software procedures) for the device being connected.

The iRMX 86 Operating System provides you with drivers for 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 Disk Controller
- iSBC 254 Bubble Memory Board
- Byte Bucket (A pseudo device to which information can only be written. Any attempt to read information from this device will result in zero bytes being returned.)
- RS232-Based Terminal or Teletypewriter

If you want to use any of these drivers in your application, refer to the iRMX 86 CONFIGURATION GUIDE. It contains detailed instructions for including specific drivers in your application system.

If you need drivers for other devices, you must write the drivers. For specific instructions refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM.

If you want more specific information about the relationship between devices, device drivers, and the iRMX 86 I/O Systems, refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

DEVICE INDEPENDENCE

The iRMX 86 Extended I/O System provides you with one set of system calls that can be used with any collection of devices. For instance, rather than using a TYPE system call for output to a terminal and a PRINT system call for output to a line printer, you may use a WRITE system call for output to any device.

This notion of one set of system calls for I/O to any collection of devices is called device independence, and it provides your application with a lot of flexibility. For example, suppose that your application logs events as they occur. The device independence of the Extended I/O System allows you to create an application that can log the events on any device rather than on just one. When the application is running and circumstances force an operator to reroute logging from the teletypewriter to the line printer or disk, your application can easily comply.

For a more detailed explanation of device independence, refer to the INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM.

THREE DISTINCT KINDS OF FILES

Although all files provided by the iRMX 86 Extended I/O System are byte (as opposed to record) oriented, the System provides three different kinds of files.

NAMED FILES

Named files are intended for use with random-access, secondary-storage devices such as disks, diskettes, and bubble memories. Named files allow your application to organize its files into a tree-like, hierarchical structure that reflects the relationships between the files and the application. Furthermore, only named files allow your application to store more than one file on a device, and only named files provide your application with access control. Named files also provide you with a good foundation for building custom access methods such as ISAM (indexed sequential access method).

For more detailed information regarding named files, read the Chapter 5 of this manual.

PHYSICAL FILES

Physical files differ from named files in that physical files allow your application more direct control over a device. Each physical file occupies an entire device, and applications can deal with the file as though it were a string of bytes. However, physical files do not provide access control.

This more-basic relationship with a device provides your application with flexibility. For example, your application can use a physical file to interpret volumes created on other systems.

Physical files also provide your application with the ability to communicate with devices that do not need the power of named files. Several examples of such devices are line printers, display tubes, plotters, and robots.

For more detailed information about physical files, read Chapter 6 of this manual.

STREAM FILES

Stream files provide a means for two tasks to communicate with each other. One task writes into the file while the other task concurrently reads from it. Stream files use no devices and provide no access control.

For more detailed information about stream files, read Chapter 7 of this manual.

FILE INDEPENDENCE

Although the Extended I/O System does support three kinds of files, almost all of the reading and writing system calls are independent of the kind of file. This allows you to create tasks and applications that can be readily switched from one kind of file to another.

For example, your application might involve two tasks that must communicate by using a stream file. In the process of developing the application system, you might implement the writing task before you implement the reading task. For the purpose of debugging the writing task, you can use a named file on a disk so you can examine the information being written. Later, after you implement the reading task, you can route the information to the stream file rather than the disk. This change does not require modifying any code in the writing task.

SEPARATION OF FILE LOOKUP AND FILE OPEN OPERATIONS

Many operating systems waste valuable time by looking up a file whenever an application tries to open one. The iRMX 86 Extended I/O System avoids this by using a special type of iRMX 86 object (called a file connection) to represent the bond between the file and an application program.

Whenever your application software creates a file, the iRMX 86 Extended I/O System returns a file connection. Your application can then use the connection to open the file without suffering the expense of having the Extended I/O System look up the file. Even when your application opens an existing file, the application can present the file connection and bypass the file-lookup process.

File connections provide a second benefit, one that relates to access control. Any connection to a named file embodies the access rights to the file. This means that the Extended I/O System computes access only once (when the file connection is created), rather than each time the file is opened.

Yet another benefit of file connections is that several of them can simultaneously exist for the same file. This allows several tasks to concurrently access different locations in the file. This is possible because each file connection maintains a pointer to keep track of the location within the file where the task is reading or writing.

The process of obtaining a file connection is discussed in each of Chapters 5, 6, and 7 of this manual.

FILE SHARING AND ACCESS CONTROL

The iRMX 86 Extended I/O System provides your application with the ability to share files and, in the case of named files, to control access to files.

FILE SHARING

In a multitasking system, it is often useful to have several tasks manipulating a file simultaneously. For example, consider a transaction processing system in which a large number of operators concurrently

manipulate a common data base. If each terminal is driven by a distinct task, the only way to implement an efficient transaction system is to have the tasks share access to the data-base file. The iRMX 86 Extended I/O system allows multiple tasks to concurrently access the same file.

For more detailed information about sharing files, refer to the Chapters 5, 6, and 7 of this manual.

ACCESS CONTROL

Also useful in a multitasking system is the ability to control access to a file. For instance, suppose that several engineering departments share a computer. An engineer in one department may want to control access to her files as follows:

- Allow herself the ability to read, write, and delete her files.
- Allow other engineers in her department to read and write the files, but deny them permission to delete the files.
- Allow engineers of other departments to only read the files.

Named files provide your application with precisely this kind of access control.

For more detailed information regarding access control, read Chapter 5 of this manual.

BUFFERING WITH OVERLAPPED I/O

The iRMX 86 Extended I/O System provides buffering and overlapping of I/O operations.

ADVANTAGES OF USING BUFFERS

Whenever one of your application programs opens a connection, the program must specify the number of buffers to be provided by the Extended I/O System. The number of buffers that your program requests directly affects the behavior of the Extended I/O System as it reads and writes information through the connection. Specifically:

- Zero Buffers

  The Extended I/O System will actually access the file each time your application invokes a read system call or a write system call. For example, if your application code asks the Extended I/O System to read 30 bytes, the Extended I/O System will access the file and read exactly 30 bytes. If the file resides on a physical device, such as a disk, the Extended I/O System will access the file for each read or write request.

- One Buffer

  The Extended I/O System will read and write information one buffer at a time. For instance, when your application program asks the Extended I/O System to read 30 bytes, the System will instead read enough information to fill the entire buffer. Using this method, the Extended I/O System might be able to satisfy several additional requests without actually reading the file.

  This method of transferring a full buffer at a time is called blocking. Blocking can significantly improve the performance of an application system by reducing the number of times that the Extended I/O System must actually transfer information to or from a file on a device. In general, blocking is more valuable in sequential I/O than in random I/O.

- Two or More Buffers

  If your application requests two or more buffers, the Extended I/O System can use blocking and can overlap I/O operations by using read-ahead and write-behind algorithms. Reading ahead and writing behind are techniques for allowing tasks of your application system to continue running while the Extended I/O System is transferring information to or from devices.

  Reading ahead is particularly useful when your application is performing sequential (rather than random-access) I/O. This arises from the Extended I/O System's ability to more accurately predict during sequential reading the location of the next data to be required by the application.

  Writing behind is also more suited to sequential I/O than to random-access I/O.

BUFFER SIZE

If you are responsible for configuring your application system, you should be aware that buffer sizes are, at least partially, a function of some parameters that you set during the configuration process. The next few paragraphs discuss these parameters. However, if you are not involved with configuration of your system, you can skip over these paragraphs without missing any crucial information.

When your application requests one or more buffers, the Extended I/O System computes the size of the buffers as a function of two configuration parameters -- the granularity of the device, and the suggested buffer size for the device. The granularity is a Basic I/O System configuration parameter, and the suggested buffer size is an Extended I/O System configuration parameter. Refer to the iRMX 86 CONFIGURATION GUIDE for detailed information about these two parameters.

When your application program opens a connection, the Extended I/O System creates buffers equal to the largest integral multiple of the device granularity that does not exceed the suggested buffer size. There are two exceptions to this rule:

- If the device granularity is zero, the Extended I/O System will create buffers equal to the suggested buffer size.
- If the device granularity is greater than the suggested buffer size, the Extended I/O System will create buffers equal to the suggested buffer size.

LOGICAL NAMES FOR FILES AND DEVICES

The Extended I/O System allows your application program to use logical names to refer to files and devices. A logical name is a string of characters that the Extended I/O System associates with a particular file connection or device connection. (A device connection relates to devices in the same way that a file connection relates to files. Refer to the Chapter 4 of this manual for a precise definition.)

The Extended I/O System implements logical names by using object directories to catalog a connection under the associated logical name. But rather than using a single object directory and consequently restricting the scope of the logical name to one job, the Extended I/O System uses three object directories to supply a choice of three scopes:

- A task can catalog a logical name in the directory of its own job. The scope of the logical name is then restricted to the tasks contained in the same job.
- A task can catalog a logical name in the directory of the system's root job. This provides a universal scope because any task in the system can lookup a logical name defined in the root job's object directory.
- A task can catalog a logical name in a third object directory that has more scope than the local job, but less than the root job. The job owning this third object directory is called the global job. (To specify a specific job as being the global job for your application job, use the object directory of your application job. Catalog a token for the global job under the logical name RQGLOBAL.)

Whenever one of your tasks asks the Extended I/O System to lookup a logical name, the System will first search the object directory of the local job. If the logical name is not defined there, the Extended I/O System will search the object directory of the global job and, if necessary, the root job. As soon as the Extended I/O System finds a definition for the logical name, the system will stop searching and return the connection found.

By providing this progressively more global search, the Extended I/O System allows you to take advantage of the three degrees of scope that you can provide your logical names:

- If you wish to share connections with tasks only in the local job, catalog the logical name in the local object directory.
- If you wish to share a connection with all tasks in the system, catalog the logical name in the root job's directory.
- If you wish to allow tasks of several specific jobs to share logical names, designate one global job for all of the jobs, and catalog the logical names in the object directory of the global job.

AUTOMATIC REATTACHMENT TO REMOVED MEDIA

If your application uses the Extended I/O System, and an operator removes media (such as disks or diskettes) from a device, the Extended I/O System monitors the status of the device. When the operator replaces the removed media, the Extended I/O System automatically reattaches the device as soon as it is accessed, making it available to the tasks of your application system.

CHAPTER 4. EXTENDED I/O SYSTEM TERMINOLOGY

There are several concepts that you must understand if you wish to use the iRMX 86 Extended I/O System. These concepts are:

- system programmers
- devices
- volumes
- files
- connections
- logical names
- I/O jobs
- path$ptr parameters and default prefixes

The following sections explain these concepts.

SYSTEM PROGRAMMERS

There are two programming roles associated with the iRMX 86 Operating System. One role involves using system calls and objects that affect only your own job, while the other role involves manipulating system resources and characteristics. These two roles are called application programming and system programming.

Although the roles have different names, separate people are not required. One individual can perform both roles. The reason for the distinction is that the actions of the system programmer affect the performance and security of the entire system, whereas the actions of the application programmer have a more limited effect.

At several locations in this manual you will find mention of programs written by system programmers. Because of the broad effect of these programs, they are only briefly described in this manual. For more detailed information you must refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

DEVICES

The iRMX 86 notion of a device probably corresponds to what you are familiar with. Flexible diskette drives, line printers, magnetic tape drives, and hard disk drives are all examples of devices.

However, there are two situations where the iRMX 86 notion of device may differ from yours:

- Several Machines on One Controller

  Even if several machines are governed by one controller, the Extended I/O System considers each machine to be a distinct device.

- Several Platters on One Spindle

  Generally, when several platters reside on a single spindle, the Extended I/O System considers the entire spindle to be one device. The exception to this arises when one platter is removable and the others are fixed. In such cases the removable platter is deemed a different device than the fixed platters, and the fixed platters all constitute one device.

VOLUMES

A volume is the actual medium used to store the device's information. If the device is a flexible disk drive, the volume is a diskette. If the device is a magnetic tape drive, the volume is the reel of tape. If the device is a multiplatter hard disk drive, the volume is the disk pack.

FILES

Some operating systems consider a file to be a device, while others consider a file to be the information stored on a device. The Extended I/O System considers a file to be information.

The Extended I/O System provides three kinds of files, each of which have characteristics making it unique. These characteristics are described in general terms in Chapter 3 and in detailed terms in Chapters 5, 6, and 7.

Regardless of the kind of file, the Extended I/O System presents information to applications in the form of a byte string rather than in records.

CONNECTIONS

A connection is an iRMX 86 object that can represent either of two things:

- The bond between iRMX 86 jobs and devices
- The bond between iRMX 86 jobs and files

DEVICE CONNECTIONS

Before the tasks of your application can manipulate files on a particular device, the device must be attached. (Because this process is typically performed by system programmers rather than application programmers, it is discussed in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.) When a program successfully attaches a device, the Extended I/O System creates a connection (a type of iRMX 86 object) that includes information describing the attached device. Such connections are called device connections.

Tasks typically obtain device connections by invoking a system program written by your system programmer. The system program attaches the device and catalogs the connection under a logical name in the object directory of the root job. Then your application program can use the logical name when invoking other system calls.

NOTE

A device cannot be multiply attached. In other words, at any one time no more than one device connection can exist for each device. However, once a device is detached, it can be reattached.

FILE CONNECTIONS

Whenever your application creates or attaches a file, the Extended I/O System returns a connection that represents the bond between the application (iRMX 86 job) and the file. This kind of connection is called a file connection.

NOTE

Files can be multiply attached. In other words, more than one connection can exist simultaneously for any file.

The reason for distinguishing between the file and the file-to-application bond is so several tasks can concurrently use the file. To support this sharing of files, file connections provide the Extended I/O System with information describing the bond. This information includes:

- a file pointer

  This is a number that tells the Extended I/O System where within the file to read, write, or truncate. The Extended I/O System automatically maintains this pointer as your task reads and writes sequentially. However, if your task must use random access, the task can modify this number by using the S$SEEK system call.

- an open-mode indicator

  The Extended I/O System sets this variable when your task calls the S$OPEN system call to open this connection to the file. The open-mode indicator tells the Extended I/O System how your task is going to use the connection. This variable can assume any of four values: open for read, open for write, open for read and write, and not open.

NOTE

There is one restriction regarding the sharing of file connections. If a task in one job obtains a file connection that was created in a different job, the task cannot successfully use the connection to perform I/O operations. However, the task can catalog the connection under a logical name, and use the logical name in the ATTACH$FILE system call to obtain a second connection that can be used without restriction.

LOGICAL NAMES

The Extended I/O System allows your tasks to use symbolic names for file and device connections. These symbolic names are known as logical names.

LOGICAL NAMES FOR DEVICE CONNECTIONS

Whenever a task uses the Extended I/O System to attach a device, the task can specify a logical name. After generating the device connection, the Extended I/O System automatically catalogs it (under the logical name provided by the task) in the object directory of the root job. Then any task in the system can obtain the device connection by looking up the logical name or by using the logical name in the path$ptr parameter of a system call.

LOGICAL NAMES FOR FILE CONNECTIONS

Similarly, when one of your tasks obtains a connection to file, the task (rather than the Extended I/O System) can catalog the connection in an object directory under a logical name, such as DATABASE. Once the file connection has been cataloged, other tasks in the system can can use it in either of two ways. They can use the LOOKUPCONNECTION system call to obtain the connection, or they can use the logical name in the path$ptr parameter of a system call.

SYNTAX FOR LOGICAL NAMES

To define a logical name you must use an iRMX 86 STRING (see Appendix A of this manual for a definition of iRMX 86 STRING). The STRING must contain 12 or fewer ASCII characters that obey the following rules:

- The hexadecimal representation of each character must be between 020h and 07Fh.
- None of the characters may be a colon (:), slash (/), an up-arrow (↑) , or a circumflex (^).

INTERPRETATION OF LOGICAL NAMES

The Extended I/O System adheres to the following three rules when interpreting logical names:

- No distinction is made between uppercase letters and lowercase letters. For example, xyz and XYZ are considered to be the same logical name.
- Leading and embedded blanks are significant. For example, if the STRING used to define a logical name contains a leading blank followed by two Xs, the logical name is not simply two Xs. Rather it is a blank followed by two Xs.
- When the Extended I/O System looks up a logical name, it checks as many as three object directories. As soon as the Extended I/O System finds a definition for the logical name, the System terminates the search and uses the connection associated with the logical name.

  The three object directories examined by the Extended I/O System are (in order of checking):

  - The object directory of the local job. The local job is the job that contains the task that caused the Extended I/O System to lookup the logical name.

- The object directory of the global job. The notion of a global job is closely related to the notion of an I/O job. Refer to the "I/O Jobs" section of this chapter for a definition of global job.

- The object directory of the root job. The root job is the job that is created when your system is first started. For a detailed definition, refer to the iRMX 86 CONFIGURATION GUIDE.

I/O JOBS

Any job using the system calls provided by the Extended I/O System must conform to certain requirements. Any job that meets these requirements is called an I/O job. These requirements include (but are not limited to):

- Cataloging an entry in the job's object directory under the name RQGLOBAL. This entry, which must refer to a job, tells the Extended I/O System which job to use as the global job. The notion of global job relates to the process of looking up logical names. Refer to Chapter 3 of this manual for more information regarding global jobs.

- Cataloging an entry in the job's object directory under the name $. This entry, which must refer to a device connection or a file connection, specifies the default prefix for the I/O job. The notion of default prefix is discussed in the "Path$ptr Parameters and Prefixes" section of this chapter.

- Cataloging an entry in the job's object directory under the name R?USER. This entry, which must refer to a user object, specifes the default user for the I/O job. The notions of users and default users are thoroughly discussed in Chapter 5 of this manual.

The purpose of an I/O job (as opposed to a non-I/O job) is to provide an environment in which tasks can invoke the system calls provided by the Extended I/O System. In fact, if a task that is not in an I/O job attempts to invoke a system call provided by the Extended I/O System, the result is very likely to be an exception code.

CREATING I/O JOBS

There are two ways to create an I/O job. One way involves using the CREATEIOJOB system call provided by the Extended I/O System. And the other way is to tell the Extended I/O System, during the process of configuration, to create one for you.

Creating an I/O Job by Using CREATEIOJOB

If you use the CREATEIOJOB system call to create an I/O job, the Extended I/O System ensures that the resultant job meets all the requirements of an I/O job. The Extended I/O System automatically initializes the new job's object directory to provide the environment required to invoke system calls of the Extended I/O System.

However, there is one restriction. The task that invokes the CREATEIOJOB system call must be running within an I/O job. This restriction leads to an obvious question. How is the first I/O job created? The answer is to create the first I/O job during the configuration of your application system.

Creating an I/O Job During Configuration of Your System

During the process of configuring your application system, you can ask the Extended I/O System to create an I/O job. The actual creation of the job will not take place until the root job initializes the system. For more information about this technique, refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that describes the configuration of the Extended I/O System.

SYSTEM CALLS RELATING TO I/O JOBS

There are two system calls that relate to I/O jobs. The CREATEIOJOB and EXITIOJOB system calls are both described in Chapter 8 of this manual.

PATH$PTR PARAMETERS AND DEFAULT PREFIXES

Some of the system calls provided by the Extended I/O System refer to files rather than to connections. All such calls require a path$ptr parameter to identify the file to be attached, created, or otherwise manipulated. And, depending upon the value of the path$ptr parameters, these system calls can also use the default prefix of the calling task's job to determine which file to manipulate.

PATH$PTR PARAMETERS

The complete interpretation of the path$ptr parameter depends upon the kind of file (named, physical, or stream) being manipulated. Consequently, the detailed interpretation of this parameter is discussed in Chapter 5 for named files, Chapter 6 for physical files, Chapter 7 for stream files, and in Chapter 8 for each system call that requires a path$ptr parameter.

However, one aspect of the path$ptr parameter applies to all three kinds of files. If the parameter is set to zero, or if it points to a null STRING (an iRMX 86 STRING containing zero characters), the Extended I/O System will manipulate the file indicated by the default prefix of the calling task's job.

DEFAULT PREFIX

The default prefix is an attribute of an I/O job. Specifically, it is a connection (either a device connection or a file connection) that is cataloged under the name $ in the object directory of the job. Whenever any task of the job invokes a system call but fails to specify which file is to be manipulated, the Extended I/O System looks up the default prefix and manipulates the associated connection.

CHAPTER 5. NAMED FILES

Named files are intended for use with random-access, secondary storage devices such as disks, diskettes, and bubble memories. Named files provide several features that are not provided by physical or stream files. These features include:

- Multiple Files on a Single Device
- Hierarchical Naming of Files
- Access Control

These features combine to make named files extremely useful in systems that support more than one application and in applications that require more than one file.

MULTIPLE FILES ON A SINGLE DEVICE

As shown in Figure 5-1, your application can use named files to implement more than one file on a single device. This can be very useful in applications requiring more than one operator, such as transaction processing systems.

HIERARCHICAL NAMING OF FILES

The iRMX 86 named files feature allows your application to organize its files into a number of tree-like structures as depicted in Figure 5-1. Each such structure, called a file tree, must be contained on a single device, and no two file trees can share a device. In other words, if a device contains any named files, the device contains exactly one file tree. Also, named file trees must fit on a single volume.

Each file tree consists of two categories of files -- data files and directories. Data files (shown as triangles in Figure 5-1) contain the information that your application manipulates, such as inventories, accounts payable, transactions, text, source code, or object code. In contrast, directory files (shown as rectangles) contain only pointers to other files. The purpose of the directory files is to provide you with a large degree of flexibility in organizing your file structure.

To illustrate this flexibility, take a close look at Figure 5-1. This figure shows how named files can be useful in multi-user systems. The figure is based on a collection of hypothetical engineers who work for three departments (Departments 1, 2 and 3). Each engineer is responsible for his own files.

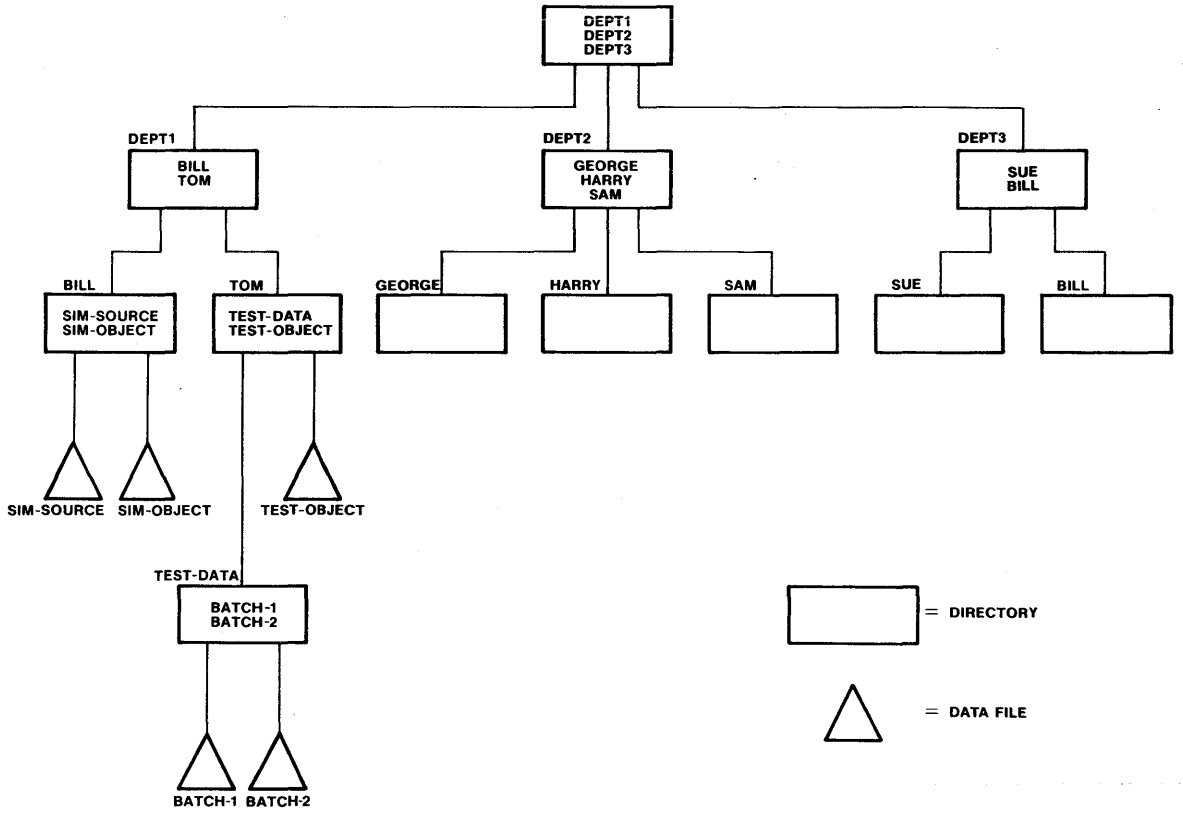


Figure 5-1. Example of a Named-File Tree

This multiperson organization is reflected in the file tree. The uppermost directory (called the device's root directory) points to three "department directories." Each department directory points to several "engineer's directories." The engineers can organize their files as they wish by using their own directories.

Each file (directory or data) has a unique shortest path connecting it to the root directory of the device. For instance, in Figure 5-1, the file called SIM_SOURCE has the path DEPT_1/BILL/SIM_SOURCE, where the slash (/) is used to separate the components of the the path. This notion of "path" reflects the hierarchical nature of the named-file tree.

Another characteristic of hierarchical file naming is that there is less chance for duplicate file names. For example, note that Figure 5-1 contains directories for two individuals named Bill. (These directories are on the extreme left and right of the third level of the figure.) Even if the rightmost Bill had a data file with the file name of SIM_OBJECT, its path would differ from that leftmost Bill's SIM_OBJECT. Specifically, the leftmost SIM_OBJECT is identified by

DEPT_1/BILL/SIM_OBJECT

whereas the rightmost SIM_OBJECT would be identified by

DEPT_3/BILL/SIM_OBJECT

Now that you know what a named file is, let's look at how the tasks of your application tell the Extended I/O System which named file to manipulate.

SYSTEM CALLS REQUIRING CONNECTIONS

Once you have a file connection for a particular named file, you can use a token for the connection as the connection parameter of any of the following system calls to perform I/O through the connection:

S$CLOSE
S$DELETE$CONNECTION
SGETCONNECTION$STATUS
S$OPEN
S$READ$MOVE
S$SEEK
S$SPECIAL
S$TRUNCATE
S$WRITE$MOVE

However, if the connection was created by a task in a different job, your task should not use the connection in any of these system calls. Rather, your task should first obtain a new connection to the same file by performing the following steps:

1. Catalog the current connection in the object directory of your task's job. This establishes a logical name for the current connection.

2. Using the newly defined logical name, invoke the S$ATTACH$FILE system call to obtain another connection to the same file. Your task can use this second connection to invoke any of the system calls listed above.

If your task does attempt to use a connection created in another job, the Extended I/O System will return an exception code rather than performing the requested function.

SYSTEM CALLS REQUIRING PATHS

In order to use any of the following system calls, your tasks must use an Extended I/O System path, rather than a connection, to tell the Extended I/O System which file you wish to manipulate:

```
S$ATTACH$FILE
S$CHANGE$ACCESS
S$CREATE$DIRECTORY
S$CREATE$FILE
S$DELETE$FILE
S$GET$FILE$STATUS
S$RENAME$FILE
```

For named files, an Extended I/O System path has two components. The first component is called a prefix, and the second is called the subpath. Let's examine these components one at a time.

Prefixes

A prefix is a logical name for a connection to either a device, a named directory file, or a named data file. The purpose of the prefix is to tell the Extended I/O System where to begin interpreting the subpath. Let's look at each of the possible interpretations that the Extended I/O System can derive from a prefix:

- If the prefix is a connection to a device, the Extended I/O System will begin scanning the subpath at the root directory of the device.

- If the prefix is a connection to a named directory file, the Extended I/O System will begin scanning the subpath at the specified directory.

- If the prefix is a connection to a named data file, the Extended I/O System will check to see if the subpath is null. If it is, the Extended I/O System will manipulate the file indicated by the prefix. If the subpath is not null, the Extended I/O System will return an exception code indicating that your application program is attempting to use a data file as though it were a directory file.

Subpaths

A subpath is a sequence of directory names or a sequence of directory names followed by the name of a data file. For instance, referring to Figure 5-1, TOM/TEST_DATA/BATCH_1 is a subpath that leads from the DEPT_1 directory to the data file named BATCH_1. Another example from the same figure is TOM, which is a subpath that leads from the directory named DEPT_1 to the directory named TOM.

Using Prefixes in Conjunction with Subpaths

The tasks of your application system can use a prefix in conjunction with a subpath to create a complete path for a named file. The prefix generally refers to a directory and the subpath generally refers to a directory or data file that is a descendant of the directory indicated by the prefix.

Specifying Paths in System Calls

Those system calls that require paths have a path$ptr parameter. The tasks of your application system can use this path$ptr parameter along with the default prefix to specify the file to be manipulated.

Path Syntax

When your application tasks invoke a system call that requires a path, the tasks must provide a path$ptr parameter. When dealing with named files, this parameter is a POINTER to a STRING (see Appendix A for a definition of STRING) that must be in one of the following four forms:

- NULL STRING

  If the STRING is zero characters long, the Extended I/O System will act on the file indicated by the default prefix of the calling task's job.

- LOGICAL NAME ONLY

  If the STRING consists only of a logical name enclosed in colons, the Extended I/O System will look up the logical name and obtain the associated connection. Then, because the subpath is empty, the Extended I/O System will act on the data file or directory file indicated by the connection.

- SUBPATH ONLY

  The STRING can consist of a subpath without a prefix. The Extended I/O System interprets such subpaths by starting at the directory indicated by the default prefix of the calling task's job. Then the Extended I/O System follows the subpath from directory to directory until it reaches the final component of the subpath. This final component is the file on which Extended I/O System will act.

  Be aware that whenever the STRING contains a subpath without a logical name, the default prefix must be a connection to a device or to a named directory file. If, instead, the default prefix is a connection to a named data file, the Extended I/O System will return an exception code indicating that your task is attempting to use a data file as a directory.

The following subpath is an example of the most common form:

```
A/B/C/D
```

where A, B, and C are the names of directory files, and D is the name of either a directory or data file. This example causes the Extended I/O System to start at the default directory and descend to Directories A, B, and C in order. Then it acts on D.

An example of a less common form of subpath is:

```
↑A/B/C/D
```

where the up-arrow (↑) or circumflex (^) tells the Extended I/O System to ascend one level in the heirarchy of files. In other words, the Extended I/O System would read this example as: "Start with the directory indicated by the default prefix and ascend to its parent. Then descend to directories A, B, and C in order. Then act on File D."

The Extended I/O System can also accept consecutive up-arrows. For example,

```
↑↑A/B/C
```

would cause the Extended I/O System to start with the directory indicated by the default prefix and ascend two levels before interpreting the remainder of the subpath.

Another possiblity is for the subpath to begin with a slash (/). For example,

```
/A/B/C
```

Whenever the Extended I/O System detects a slash at the beginning of a subpath, the Extended I/O System will start interpreting the remainder of the subpath at the root directory of the device indicated by the prefix.

- LOGICAL NAME FOLLOWED BY SUBPATH

Your application code can use a STRING consisting of a logical name (enclosed in colons) followed immediately by a subpath. For example,

```
:F0:A/B/C/D
```

The Extended I/O System interprets this example as follows. First, it looks up the logical name F0 in the object directory of the local job, or if necessary, the global or root job. Then it follows the subpath from the directory associated with the connection. So in the example, the Extended I/O System would find the directory associated with F0, and it would step through Directories A, B, and C. Finally, the Extended I/O System would act on File D.

USERS AND ACCESS RIGHTS

Named files provide the tasks of your application with the ability to control access to files. This ability is based on the concept of users and the concept of access rights.

USERS AND USER OBJECTS

The iRMX 86 Extended I/O System implements an iRMX 86 object type called a user object. Before you can find user objects useful, you must understand the concepts of user, group, and World.

Concept of User

The concept of user correlates file access to people or to iRMX 86 jobs, but the precise definition depends upon the nature of your application. For instance, if your application interfaces with several people who manipulate a data base, you might want to consider each person (or small groups of persons) a user. This would allow each individual (or small group) to maintain access different from other individuals (or other small groups).

Alternatively, if your application interfaces with only one (or even no) person, then you might wish to consider each iRMX 86 job as a user. This would allow your application to control which job accesses which file.

In more general terms, the set of entities that manipulate named files in your system is the set of all users. If you want all of these entities to be able to access any file, you can consider them to be a single user. However, if you want to distribute different access to different collections of these entities, you must divide the entities into subsets, and each of these subsets is a user.

Now let's look at an example derived from Figure 5-1. As mentioned earlier, each engineer in the figure is responsible for his own files. If an engineer wants to have unique access to his files, access different than anybody else's, the engineer is a user. On the other hand, if all engineers are willing to give uniform access to each member of his department (including himself) then the departments are the users.

Concept of Group

Closely related to the concept of user is the concept of group. A group is a collection of users who share some access. For example, suppose that each engineer in Figure 5-1 wishes to reserve for himself certain access to his own files, while allowing members of the same department different access to the same file. This can be accomplished by considering each engineer as a user, and each department as a group that includes all of the engineers in the department. By doing this, an engineer can assign himself one kind of access and his department another.

Concept of World

The concepts of user and group can be used to assign various kinds of access to different collections of users, but once in a while it is useful to assign one kind of access to all users. In order to do this, your application must employ a special group, called World, that includes all users. For example, one of the engineers in Figure 5-1 could use the World group to allow all of the other engineers to access a file.

User Objects

The Extended I/O System supports an iRMX 86 object type, called a user object, that lets you bind users to groups, including the special group called World. Whenever an application (in the form of a task within an I/O job) attempts to gain access to or create a named file, the Extended I/O System examines the application's user object to compute the kind of access permitted the application.

In effect, user objects serve a purpose analogous to that of the plastic cards that allow people to deal with automated bank tellers. If you don't have a valid plastic card, you can't use the automated teller. Similarly, if your application doesn't have the correct user object, it can't access certain named files.

User objects consist of a collection of identity codes (id's) that are assigned by your application software. The first id is the id of the user whom the object represents, and any additional id's specify groups to which the user belongs. For example, the id list of a user object might look like this:

```
0231
A4D5                (All numbers in the list are hexadecimal.)
FFFF
```

Suppose that this is the id list for the user object of Harry in Figure 5-1. Harry's id is 0231, and the other id's represent groups to which Harry belongs. For example, A4D5 could be the id representing Department 2. A group does not need a user object unless the group (rather than the users in the group) is going to create or access files.

Take a special note of the third id on the list. By convention, FFFF is the id used for the World. If you wish to take advantage of this useful convention, you must ensure that every user is considered to be part of the world. In other words, whenever you create any user object, you should include FFFF as a group to which the user belongs.

Futhermore, if you wish to allow the World to create and access files, you must create a user object for the world. The id list of the World's user object should contain a single id, FFFF. The use of this special user object is described later in this chapter, in the "Granting Access to Other Users" section.

The Extended I/O System computes access based on user objects and a file's access list. This computation is fully explained in the "Access Rights" section of this chapter.

Creating, Deleting, and Inspecting User Objects

The process of creating and deleting user objects is generally performed by system programs rather than by application programs. For example, application programs requiring user objects can invoke a system program that creates the object and passes it back to the application program through a mailbox. Another alternative that is particularly useful in systems that interact with more than one person is to create a log-on facility that creates user objects as operators enter a password.

The Basic I/O System (rather than the Extended I/O System) provides three system calls for creating, deleting and inspecting user objects. These calls, which are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, are:

```
CREATE$USER
DELETE$USER
INSPECT$USER
```

Default Users

Generally, most of the I/O operations performed within a particular iRMX 86 I/O job are performed on behalf of one user object. Recognizing this, the Extended I/O System allows your application to designate one default user per I/O job. Whenever your application invokes an Extended I/O System call that requires access checking, the Extended I/O System will verify that the default user for the I/O job has the required access.

The notion of a default user provides two benefits. First, it allows you to avoid some repetitive coding. Second, and more significantly, it allows your application to easily parameterize the user for whom I/O is being performed. For example, if your application includes an I/O job that modifies a named file on behalf of other jobs in the system, the invoking job can set the default user of the I/O job to a specific user object. Then, all of the I/O system calls will be performed on behalf of the default user.

To establish, de-establish, or ascertain the default user of a specific job, you can use two system calls provided by the Basic I/O System (rather than the Extended I/O System). The GET$DEFAULT$USER and SET$DEFAULT$USER calls are both described in the system call chapter of the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

ACCESS RIGHTS

For each named file (directory or data file), the Extended I/O System maintains a list of ordered pairs having the form (id, access rights). The id portion is the identity code for a user or a group. The access rights portion is encoded as a hexadecimal number that indicates all the access rights for the associated id. The list of pairs is called the file's access list, and the I/O System supports as many as three entries for each named file.

The kinds of access rights that a user or group can have depend on whether the file is a data file or a directory file. The kinds of access rights available for data files are:

Delete	The ability to delete the file with S$DELETE$FILE and rename the file with S$RENAME$FILE.
Read	The ability to read the file with S$READ$MOVE.
Append	The ability to add information to the end of the file with S$WRITE$MOVE.
Update	The ability to change information in the file with S$WRITE$MOVE or drop information from the end of the file with S$TRUNCATE$FILE.

The kinds of access rights available for directory files are:

Delete	The ability to delete the directory file with S$DELETE$FILE. Also allows changing the name of the directory by using S$RENAME$FILE.
Display	The ability to obtain the contents of directory files with S$READ$MOVE.
Add Entry	The ability to add files to the directory with S$CREATE$FILE, S$CREATE$DIRECTORY, or S$RENAME$FILE. This does not include permission to change existing entries.
Change Entry	The ability to change the access rights of the files in the directory with S$CHANGE$ACCESS. This does not include permission to add new entries.

The numeric values associated with the access rights are explained in the descriptions of S$CREATE$FILE and S$CREATE$DIRECTORY in the system call chapter of this manual.

When an application task creates a named file, the task uses the S$CREATE$FILE or S$CREATE$DIRECTORY system call. When the Extended I/O System actually builds the file, it initializes the access list with a single entry consisting of the id of the default user for the task's I/O job. The Extended I/O System grants this user full access to the file.

(Full access for directory files is delete, display, add entry, and change entry. For data files, full access is delete, read, append, and update.) The user who creates a file is called the owner of the file.

NOTE

The owner of a file has only one advantage over other users who can access the file, but the advantage is an important one. Only a file's owner can use the S$CHANGE$ACCESS system call to modify the file's access list without being granted explicit permission to do so.

Computing Access

Whenever an application task attempts to create a connection to an existing named file, the Extended I/O System examines the access associated with the default user object for the calling task's I/O job. The Extended I/O System then scans the access list of the file and finds all entries that match any id's (user or group) in the id list of the user object. Finally, the Extended I/O System computes the access by "or"ing together the access of each matching entry.

Consider an example. Suppose that an application task attempts to establish a connection to a file having the following access list:

(D556, 0F)
(8B01, 05)
(FFFF, 02)

Now suppose that the default user object for the I/O job of the application task has an id list of

042A
8B01
FFFF

The Extended I/O System would find that the user object has two id's that match entries in the file's access list. The id's are 8B01 and FFFF, and the corresponding access rights are 05 and 02. So the Extended I/O System would compute access by "or"ing together 05 and 02, yielding access of 07. The precise interpretation of this access depends upon whether the file is a directory or a data file, as explained previously.

Time at Which Access is Computed

The iRMX 86 Extended I/O System computes access only under two circumstances. The first circumstance is the creation of a connection. Whenever an application task creates a connection to a named file (by using the S$CREATE$FILE, S$CREATE$DIRECTORY, or S$ATTACH$FILE system calls), the Extended I/O System examines the default user object of the task's I/O job. The System uses this object and the access list of the named file to compute the access, and it embeds this access in the connection object that is returned to the application.

Later, when the application task attempts to manipulate the file via the connection, the Extended I/O System uses the connection's embedded access to decide what kind of manipulation is permitted. Even if an application changes the access list of the file or the id list of the user object, the change will have absolutely no effect on the access previously embedded in the connection.

The second circumstance under which the Extended I/O System computes access arises when an application uses either the S$DELETE$FILE or the S$CHANGE$ACCESS system calls. If the system call invocation contains any subpath other than the null subpath, the Extended I/O System will compute access to the target file before performing the desired function. If access is not granted, the Extended I/O System will deny the user the ability to delete the file or change access.

If an invocation of S$DELETE$FILE or S$CHANGE$ACCESS does contain the null subpath, the I/O System will use the access associated with the prefix (whether default prefix or explicit logical name) to decide whether or not to perform the function requested in the system call.

NOTE

If a system call invocation contains a subpath other than the null subpath, the Extended I/O System checks the access only to the last file in the path and to the parent directory of the last file. It does not check the access to any other directory files specified in the path.

Granting Access to Other Users

When an application task initially creates a named file (either data file or directory) access to the file is restricted to the creating user (the owner). However, there are two ways for the owner to allow other users to access the file.

The first technique is performed after the creation of the file. The owner of the file is always entitled to change the access to the file. So by using the S$CHANGE$ACCESS system call, the owner can provide other users access.

The second technique involves a group user object (discussed earlier under the heading of "User Objects"). If, when your application task creates a file, the default user for the task's job is a group user, the group is the owner of the file. Consequently, any user in the group can use the S$CHANGE$ACCESS system call to modify the file's access list. This means that any user in the group can grant himself access to the file.

EXTENDED I/O SYSTEM CALLS FOR NAMED FILES

The Extended I/O System provides a number of system calls that relate to named files. The following sections briefly explain the purpose of each of these system calls. The descriptions are grouped by function rather than alphabetically. These descriptions are very brief. Chapter 8 of this manual contains descriptions of most of the calls, and the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL contains descriptions of the others. If any of the following descriptions do not explicitly refer to a more detailed description, you can find such a description in Chapter 8.

OBTAINING AND DELETING CONNECTIONS

The Extended I/O System provides six system calls that relate to obtaining and deleting connections.

- S$CREATE$FILE

  This call applies only to data files. Your application software must use this call to create a new data file. When an application task invokes this call, the Extended I/O System automatically adds an entry in the parent directory for this new file.

- S$CREATE$DIRECTORY

  This call applies only to directory files. When your application software needs to create a directory, the software must use this system call. The call cannot be used to obtain a connection to an existing directory. The Extended I/O System automatically adds an entry in the parent directory for this new directory.

- S$ATTACH$FILE

  This call applies to both data and directory files. Your application tasks can use this call to obtain a connection to an existing data file or directory.

- S$DELETE$CONNECTION

  This call applies to both data and directory files. Your application tasks can use this call to delete a connection to either kind of named file. This call cannot be used to delete a device connection.

- LOGICAL$ATTACH$DEVICE

  This call does not directly apply to either data or directory files. Your application software uses this call to obtain a connection to a device and to catalog the logical name for the device in the object directory of the root job. Even though this connection is a device connection, it can be used as the prefix for the root directory of the device. This call is explained in detail in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

- LOGICAL$DETACH$DEVICE

  This call does not directly apply to either data or directory files. Your application software uses this call to delete a connection to a device and remove the logical name of the device from the object directory of the root job. This system call is explained in detail in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

MANIPULATING DATA

There are six system calls that allow tasks of your application to manipulate the data that forms a file. All six can be used with data files, while only four apply to directory files. All of the calls are described in Chapter 8 of this manual. The system calls are:

- S$OPEN

  This call applies to both data and directory files. Before your application software can use any other system calls to manipulate file data, the software must open a connection to the file. This system call is the only way to open a connection.

- S$CLOSE

  This call applies to both data and directory files. After your application software has finished manipulating a file, the application can use this system call to close the file connection. Your application can elect to leave the file open, letting the Extended I/O System close it when the connection is deleted, but there are two advantages to closing connections when they are not being used.

  The first advantage is that closing a connection releases the memory associated with the connection's buffers. Once released, this memory can again be used by the Operating System.

The second advantage derives from the fact that, when a file is shared between two or more application tasks, some of the tasks can place restrictions on the manner of sharing. For instance, a task can specify sharing with writers only. By closing a connection, your application task drops any such restrictions, improving the likelihood that the file can be used by other application tasks.

- S$SEEK

  This system call applies to both data and directory files. Whenever your application software reads, writes, or truncates a file, the requested action takes place at the location specified by the connection's file pointer. The application can tell the Extended I/O System where the operation is to take place. To do this, your application task uses the S$SEEK system call to position the file pointer of the file connection. The S$SEEK system call requires that the file connection be open.

- S$READ$MOVE

  This system call applies to both data and directory files. Your application tasks can use this system call to read file data from the location indicated by the file pointer. Before using thissystem call, your application software can use the S$SEEK system call to position the file pointer. The S$READ$MOVE system call requires that the file connection be open.

  The outcome of this system call depends upon whether a data file or a directory is being read. If your application task reads a data file, the application will receive data that makes up the file. If the application reads from a directory, the application will receive data that represents the entries of the directory.

  Each entry in a directory consists of 16 bytes. The first two bytes contain a 16-bit file-descriptor number corresponding to the file descriptor number associated with the SGETFILE$STATUS system call in Chapter 8. The remaining 14 bytes are the ASCII characters making up the name of the file to which the directory entry points. (A file's name is the last component of a path.)

- S$WRITE$MOVE

  This system call applies only to data files. Your application software uses this system call to put new information in the file. Before using this call, an application task can use S$SEEK to position the file pointer to the location within the file to receive the information. The S$WRITE$MOVE system call also requires that the file connection be open.

- S$TRUNCATE$FILE

  This system call can be used only on data files. Your application software can use this call to trim information from the end of the file. To do so, the application task first can use S$SEEK to position the file pointer to the first byte to be dropped. Then the application invokes the S$TRUNCATE$FILE call to drop any bytes at or beyond the file pointer. The S$TRUNCATE$FILE system call requires that the file connection be open.

OBTAINING STATUS

There are two status-related system calls, one for connections and one for files. The calls are SGETFILE$STATUS and S$GET$CONNECTION$STATUS. Both of these calls can be used with data files and directory files. Both of these calls are described in Chapter 8.

DELETING AND RENAMING FILES

The Extended I/O System provides one system call for deleting files, and another for renaming files. Both of these calls can be used with data files and directory files. The calls are:

- S$DELETE$FILE

  Your application tasks can use this system call to delete data files and directory files. However, any attempt to delete a directory that is not empty will result in an exceptional condition.

- S$RENAME$FILE

  Your application tasks can use this system call to rename both data files and directory files. In renaming a file, an application task can move the file to any directory in the same named file tree. For example, you can rename A/B/C to be A/X/C. In effect, this example simply moves File C from Directory B to Directory X. This means that the application task can change every component of a file's path name.

CHANGING ACCESS

The Extended I/O System provides one system call to let the tasks of your application change a file's access list. This call is S$CHANGE$ACCESS, and it applies to both data files and directories. One rule governs the use of S$CHANGE$ACCESS -- only the owner of a file or a user with change entry access to the directory containing the file can change the file's access list.

PERFORMING SPECIAL FUNCTIONS

The Extended I/O System provides the S$SPECIAL system call to allow your application software to perform functions that are peculiar to a particular device. Formatting a disk is an example of such a function. For more information, refer to the S$SPECIAL section of Chapter 8.

DELETING CONNECTIONS

The Extended I/O System provides one system call to delete connections to files. This is the S$DELETE$CONNECTION system call, and it is described in Chapter 8.

USING LOGICAL NAMES

The Extended I/O System provides three system calls that relate to logical names. All three of these system calls are discussed in detail in Chapter 8 of this manual.

- S$CATALOG$CONNECTION

  This system call allows your application tasks to create a logical name by cataloging a connection in the object directory of any job that your tasks choose.

- S$LOOKUP$CONNECTION

  This system call accepts a logical name from an application task, looks up the name in the object directories of the local, global, and root jobs, and returns a token for the first connection found. In other words, this is the system call that your application software uses to go from a logical name to a connection.

- S$UNCATALOG$CONNECTION

  This system call allows your application software to delete a logical name from the object directory of any specific job.

CREATING AND DELETING I/O JOBS

The Extended I/O System provides two system calls that relate to the creation and deletion of I/O jobs. Both of these system calls are described in Chapter 8 of this manual.

- CREATEIOJOB

  This system call creates an I/O job. Aside from using this system call, the only way to create an I/O job is to request (during configuration of the Extended I/O System) that the Extended I/O System create one for you. The primary distinction between these two techniques is that the system call allows your application tasks to create I/O jobs while the system is running. The configuration-time alternative does not.

- EXITIOJOB

  This system call provides your application tasks with a convenient method for terminating an I/O job and informing the parent job of the termination.

BASIC I/O AND NUCLEUS SYSTEM CALLS

Although the purpose of this manual is to describe the Extended I/O System, there are several system calls provided by the Basic I/O System and the Nucleus that warrant discussion here. These calls relate to user objects and to default prefixes.

USER OBJECTS

The Basic I/O System provides five calls directly related to user objects. The calls are:

- CREATE$USER

  This call is used to create a user object. Since this call is generally invoked only by system programs, it is described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

- DELETE$USER

  This call is used to delete a user object. Since this call is generally invoked only by system programs, it is described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

- INSPECT$USER

  This call is used to ascertain a user object's id and to find out to which groups the user belongs. Since this call is generally invoked only by system programs, it is described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

- SET$DEFAULT$USER

  Your application software can use this call to establish a default user for an iRMX 86 I/O job. This call is described in the System Call Chapter of the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

- GET$DEFAULT$USER

  Your application software can use this call to ascertain the default user for an iRMX 86 I/O job. This call is described in System Call Chapter of the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

The iRMX 86 Nucleus also provides three system calls that relate to default users. These calls are the CATALOG$OBJECT, UNCATALOG$OBJECT, and LOOKUP$OBJECT system calls. The default user for each I/O job is defined in the job's object directory under the name R?USER, and you can use these Nucleus system calls to change the object associated with this name. However, before using these system calls, refer to Appendix D of this manual. Also, if you need a description of the Nucleus system calls, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

DEFAULT PREFIXES

The Basic I/O System provides two system calls that relate to default prefixes. Both of these calls are described in detail in the System Calls Chapter of the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

- SET$DEFAULT$PREFIX

  Your application software can use this call to set the default prefix for any iRMX 86 job.

- GET$DEFAULT$PREFIX

  Your application software can use this call to ascertain the default prefix for any iRMX 86 job.

The iRMX 86 Nucleus also provides three system calls that relate to default prefixes. These calls are the CATALOG$OBJECT, UNCATALOG$OBJECT, and LOOKUP$OBJECT system calls. The default prefix for each I/O job is defined in the job's object directory under the name $, and you can use these Nucleus system calls to change the object associated with this name. However, before using these system calls, refer to Appendix D of this manual. Also, if you need a description of the Nucleus system calls, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

CHRONOLOGICAL OVERVIEW OF NAMED FILES

Although many system calls can be used with named files, the system calls cannot be used in arbitrary order. This section provides you with a sense of how the calls relate to one another.

CALLS RELATING TO USER OBJECTS

With one exception, the system calls relating to user objects are completely independent of other I/O System calls. The one exception is that your application must have a user object before it can use any system call requiring a user object.

There are five system calls relating to user objects. Of the five, GET$DEFAULT$USER and CREATE$USER can be invoked any time. The others, DELETE$USER, INSPECT$USER, and SET$DEFAULT$USER, can be invoked only after user objects exist.

CALLS RELATING TO PREFIXES

The GET$DEFAULT$PREFIX system call can be invoked at any time. In contrast, the SET$DEFAULT$PREFIX requires that a file or device connection exist.

CALLS RELATING TO STATUS

The Extended I/O System provides two system calls relating to status. The SGETFILE$STATUS system call can be used anytime after the file has been created. The SGETCONNECTION$STATUS system call can be used whenever your task has a connection to the file. The connection need not be open.

CALLS RELATING TO CHANGING ACCESS

The only system call related to changing access, S$CHANGE$ACCESS, can be invoked whenever your application's I/O job has a connection to a file or has both a default user object and a path to a file.

CALLS FOR PERFORMING DEVICE-SENSITIVE FUNCTIONS

There is only one system call that lets your application software perform I/O operations particular to a specific device, the S$SPECIAL system call. Your application's tasks can use this call any time after your application's I/O job has a device connection.

CALLS FOR RENAMING FILES

The one call for renaming a file, S$RENAME$FILE, can be used whenever your application's I/O job has a path to the file to be renamed, a user object, and a path that is to become the new path.

MOST FREQUENTLY USED SYSTEM CALLS

Figure 5-2 shows the chronological relationships between most frequently used I/O System calls. To use the figure, start with the leftmost box and follow the arrows. Any path that you can trace is a legitimate sequence of system calls. However, there are also sequences not represented in the figure.

Remember that an I/O job must have a default user object before any of its tasks can successfully use any system calls that check access.

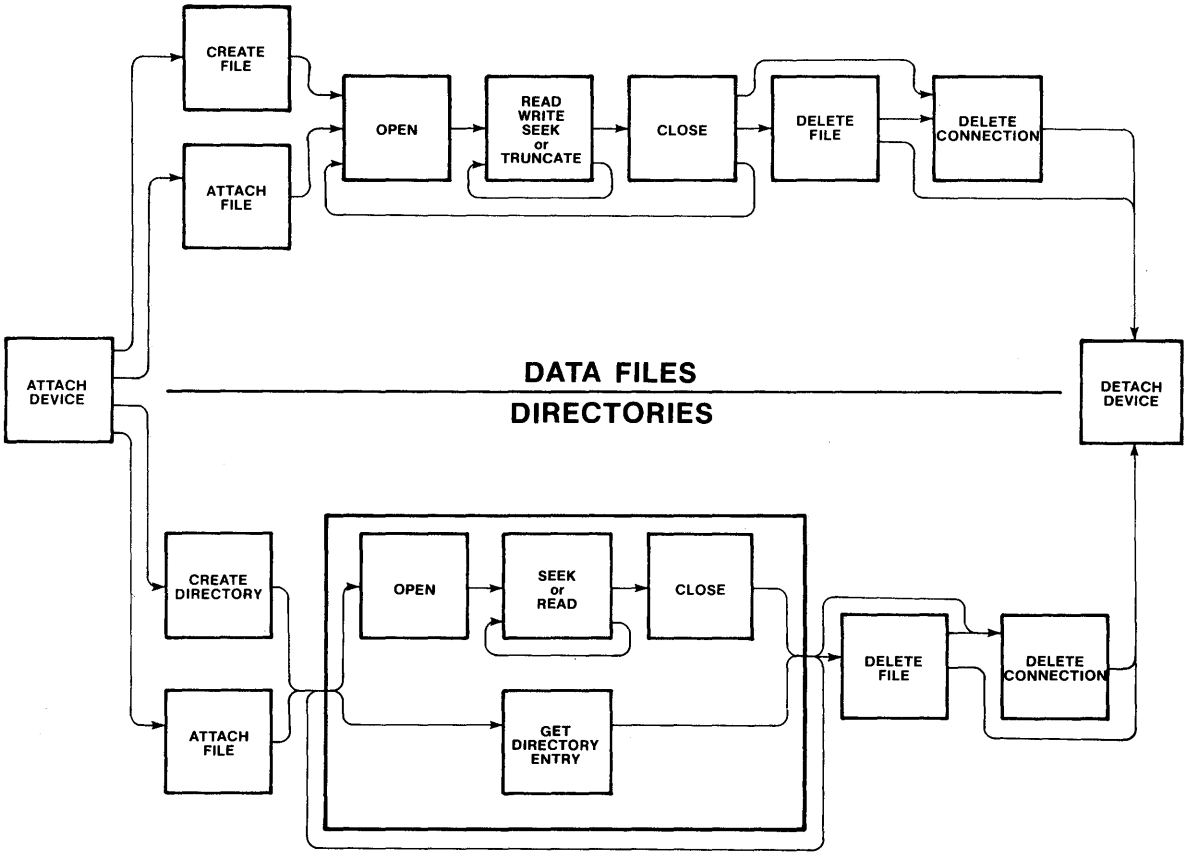


Figure 5-2. Chronology of Frequently Used System Calls for Named Files

SUGGESTION FOR MAINTAINING FILE INDEPENDENCE

If you would like the tasks of your application to be able to use any kind of file rather than only named files, you should separate the creation of the connection from the use of the connection. For instance, if your application performs I/O to a file, you should consider using two distinct tasks rather than one. The first task would be responsible for obtaining a connection, and the second task would use the connection to perform I/O. By doing this, you can design the second task to work with any kind of file.

CHAPTER 6. PHYSICAL FILES

The iRMX 86 Extended I/O System provides physical files to allow your applications to read (or write) strings of bytes from (or to) a device. In other words, a physical file occupies an entire device, and the Extended I/O System provides your applications with the ability to directly access the driver of the device.

SITUATIONS REQUIRING PHYSICAL FILES

The close relationship between a device and a physical file is particularly useful when your application system uses sequential devices. For example, you should use physical files to communicate with line printers, display tubes, plotters, magnetic tape units, and robots.

There are even some instances where you should use physical files to communicate with random devices such as disks, diskettes, and bubble memories. For instance:

- Formatting Volumes

  Whenever you create an application task to format a disk or diskette, the task must have access to every byte on the volume. Only physical files provide this kind of access.

- Using Volumes in Formats Required by Other Systems

  If your application tasks must read or write volumes that have been formatted for systems other than the iRMX 86 Operating System, you must use physical files. Your tasks will have to interpret information such as labels and file structures, but a physical file can provide your tasks with access to the raw information.

- Implementing Your Own File Format

  Suppose that your application system requires a less sophisticated file structure than that provided by iRMX 86 named files. You can build a custom file structure using a physical file as a foundation.

CONNECTIONS AND PHYSICAL FILES

Although there is a one-to-one correspondence between the bytes on a device and the bytes of a physical file, the device connection is different from the file connection. The Extended I/O System maintains this distinction to remain consistent with named files and stream files. This consistency helps you develop applications that can use any kind of file.

SUGGESTION FOR MAINTAINING FILE INDEPENDENCE

If you would like the tasks of your application to be able to use any kind of file rather than only physical files, you should separate the creation of the connection from the use of the connection. For instance, if your application performs I/O to a file, you should consider using two distinct tasks rather than one. The first task would be responsible for obtaining a connection to the file, and the second task would use the connection to perform I/O. By maintaining this separation, you can design the second task to work with any kind of file.

If you choose to use this two-task approach, be sure that both tasks are in the same job. This will eliminate the difficulties associated with passing a file connection from one job to another.

USING PHYSICAL FILES

Several system calls can be used with physical files, but the order in which they are used is not arbitrary. The following list provides a brief description (in chronological order) of what an application must do to use a physical file.

1. Obtain a device connection.

   This is necessary for two reasons. When your task creates the physical file, the device connection tells the Extended I/O System which device is to contain the file and that the file must be a physical file.

   Since the process of attaching a device is restricted to system programs, you must create a system program. This program must use the LOGICAL$ATTACH$DEVICE system call to obtain the device connection. When issuing this call, the system program must use the device name that was assigned to the device during system configuration. For instructions as to how to assign names to devices, refer to the iRMX 86 CONFIGURATION GUIDE.

   Because devices cannot be multiply attached, your system program must be written so as to call LOGICAL$ATTACH$DEVICE only once. The LOGICAL$ATTACH$DEVICE system call obtains a device connection and catalogs the connection under the logical name provided by the system program. Other tasks wishing to use the device connection can then lookup the connection by using the device's logical name.

   The LOGICAL$ATTACH$DEVICE system call is described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

2. Obtain a file connection.

   In order to obtain a file connection, your application task should use one of the following two system calls--S$CREATE$FILE, or S$ATTACH$FILE. The decision as to which system call to use depends upon your task's awareness of the existence of the file.

There are two circumstances under which your task should use the S$CREATE$FILE system call to obtain a connection. The first circumstance is when your task does not know whether the file already exists, and the second is when your task knows that the file does not yet exist.

When invoking the S$CREATE$FILE system call, set the path$ptr parameter to point to a STRING containing the logical name of the device (enclosed in colons, as in :F0:). This will tell the Extended I/O System which device you want as your physical file.

If, on the other hand, your task is certain that the file already exists, use the S$ATTACH$FILE system call to obtain the file connection. Your task can do this in either of two ways:

- It can set the path$ptr parameter of the call to point to a STRING containing the device's logical name surrounded by colons (as in :F0:).

- If the task knows a logical name for a connection to the file, it can set the path$ptr parameter of the call to point to a STRING containing the connection's logical name surrounded by colons (as in :DATABASE:).

Either way, the Extended I/O System will return a connection to the physical file.

This careful distinction between the S$CREATE$FILE and the S$ATTACH$FILE system calls is necessary to be consistent with named files. If you want your application to work with any kind of file, you must maintain this consistency.

3. Open the file connection.

   Use the S$OPEN system call to open the connection. When opening the connection, your task must specify whether the task plans to read, write, or do both using the connection. The task must also specify how many buffers the Extended I/O System is to use when reading from or writing to the file. Chapter 8 of this manual explains how to do this.

4. Manipulate the file.

   There are four system calls that can be used to read, write, or otherwise manipulate your physical file:

   - The S$READ$MOVE and S$WRITE$MOVE are used to read and write information from (to) the physical file.

   - The S$SEEK system call can be used to manipulate the file connection's file pointer if the device is a random device such as disk, diskette, or bubble. (If you are writing a device driver for a magnetic tape unit, you can design it to support S$SEEK. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM.)

- The S$SPECIAL system call can be used to request device dependent functions from the device driver. For example, your tasks can use the S$SPECIAL system call to have the Extended I/O System format a disk for use with the iRMX 86 Operating System. Be aware that use of special functions generally prevent a task from being device independent.

All four of these system calls are described in Chpater 8 of this manual.

5. Close the file connection.

   Use the S$CLOSE system call to close the connection. Note that your application can repeat steps 2, 3, and 4 any number of times. The S$CLOSE system call is described in Chapter 8 of this manual.

6. Delete the connection.

   Use the S$DELETE$CONNECTION system call to delete the connection. This is only necessary if the tasks of your application are completely finished using the file. This system call is described in Chapter 8 of this manual.

7. Request that the device be detached.

   Let the system program know when your task no longer needs the device. The system program should keep track of the number of tasks using the device and should avoid detaching it until it is no longer being used by any task. Only then should the system program use the LOGICAL$DETACH$DEVICE system call to detach the device.

   The LOGICAL$DETACH$DEVICE system call is described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

CHAPTER 7. STREAM FILES

Stream files provide a means for one task to send large amounts of information to a second task, even when the two tasks are in different jobs. Be aware that stream files are only one of several techniques for job-to-job communication. If you are not familiar with other techniques, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.

The aspect of stream files that makes them very useful is that they allow a task to communicate with a second task as though the second task were a device. This extends the notion of device independence to include tasks.

Since two tasks (called the reading task and the writing task) are involved in using each stream file, the tasks must cooperate. There are a large number of protocols that work, but the ones provided later in this chapter serve as good illustrations.

SUGGESTION FOR MAINTAINING FILE INDEPENDENCE

If you would like your reading and writing tasks to be able to use named files or physical files rather than only stream files, you should incorporate a third task into the protocol. The purpose of this third task is to perform the one part of the protocol that depends on the kind of file being used--the creation of the file.

STREAM FILE PROTOCOLS

The interaction between the tasks is divided into three protocols--one each for the creating, writing, and reading tasks. If you choose to avoid using a separate task to create the file, you can have the writing task perform the creating protocol before it performs the writing protocol. However, by eliminating the creating task, you force the writing task to require that only stream files be used. In order to allow both the reading and writing tasks to be independent of the kind of file being used, you should use a separate creating task.

The following protocols will work even if the three tasks are in different jobs. They will also work regardless of the order in which they are executed.

PROTOCOL FOR THE CREATING TASK

The creating task is responsible for obtaining a device connection for the stream file pseudo device, and for creating the stream file. It also must catalog the file connection under a logical name so the reading and writing tasks can attach the file. Remember that this task is not device independent--it works only for stream files. This protocol involves two steps:

1. Creating a stream file.

   During the process of configuring the Extended I/O System, one of the configuration parameters that must be entered is a logical name for the stream file device. During the process of starting up the system, the Extended I/O System attaches the stream file pseudo device and catalogs the device connection under the logical name. Your tasks can then use this logical name to obtain the device connection.

   For the purpose of understanding this protocol, assume that the logical name is STREAM. However, be aware of the possiblility that, in your system, the logical name might be something other than STREAM. To ascertain the locical name for your system, consult the person(s) responsible for configuring your system.

   In order to create a stream file, the creating task need only invoke the S$CREATE$FILE system call using a path$ptr parameter pointing to a STRING of the following form:

   ```
   :STREAM:
   ```

   where STREAM is the logical name for the stream file device connection. The S$CREATE$FILE system call, which is described in Chapter 8 of this manual, returns a connection to the newly created stream file.

2. Catalog the file connection under a logical name.

   The creating task should invoke the S$CATALOG$CONNECTION system call to establish a unique logical name (for example, SF23) for each specific stream file. The reading and writing tasks can then use the logical name to attach the file. The S$CATALOG$CONNECTION system call is described in Chapter 8 of this manual.

PROTOCOL FOR THE WRITING TASK

The writing task must perform five steps in order to ensure that it establishes communication with the reading task. The steps are:

1. Obtain a connection to the stream file.

   The writing task should use the logical name of the file connection (for example SF23) and invoke the S$ATTACH$FILE

system call (which is described in Chapter 8 of this manual) to obtain the file connection. To do this, the task should set the path$ptr parameter of the system call to point to a STRING containing the file connection's logical name enclosed in colons (as in :SF23:).

2. Open the file connection for writing.

   Use the S$OPEN system call to open the file connection for writing. Set the connection parameter to the token for the file connection, and set the mode parameter to write. The S$OPEN system call is described in Chapter 8 of this manual.

3. Write information to the stream file.

   Use the S$WRITE$MOVE system call as often as desired to write information to the stream file. Use the token for the file connection as the connection parameter. The S$WRITE$MOVE system call is described in Chapter 8 of this manual.

4. Close the connection.

   When finished writing to the stream file, use the S$CLOSE system call to close the connection. Note that after this step, the writing task can repeat steps 2, 3, and 4. The S$CLOSE system call is described in Chapter 8 of this manual.

5. Delete the connection.

   Use the S$DELETE$CONNECTION system call to delete the connection to the stream file. The S$DELETE$CONNECTION system call is described in Chapter 8 of this manual.

PROTOCOL FOR THE READING TASK

The reading task must perform the following seven steps to successfully read the information written by the writing task:

1. Obtain the file connection for the stream file.

   The reading task should use the file's logical name (for example, SF23) and invoke the S$ATTACH$FILE system call to obtain the file connection. To do this, the task should set the path$ptr parameter of the system call to point to a STRING containing the file connection's logical name enclosed in colons (as in :SF23:).

2. Open the file connection for reading.

   The task should use the S$OPEN system call to open the file connection for reading. Set the connection parameter to the token for the file connection, and set the mode parameter to read. The S$OPEN system call is described in Chapter 8 of this manual.

3. Read information from the stream file.

   The task should use the S$READ$MOVE system call as often as needed to read information from the stream file. Use the token for the file connection as the connection parameter. The S$READ$MOVE system call is described in Chapter 8 of this manual.

4. Close the connection.

   When finished reading from the stream file, the task should use the S$CLOSE system call to close the connection. Note that after this step, the reading task can repeat steps 2, 3, and 4. The S$CLOSE system call is described in Chapter 8 of this manual.

5. Delete the connection.

   The task should use the S$DELETE$CONNECTION system call to delete the connection to the stream file. The S$DELETE$CONNECTION system call is described in Chapter 8 of this manual.

6. Delete the file's logical name created by the creating task.

   The task should use the S$UNCATALOG$CONNECTION system call (which is described in Chapter 8 of this manual) to delete the logical name for the file. In our example, this logical name is SF23. Do not delete the logical name for the stream file device.

7. Delete the file connection created by the creating task.

   The reading task should use the S$DELETE$CONNECTION system call to delete the file connection that the creating task obtained. Once this connection is deleted, the Extended I/O System will automatically delete the stream file.

CHAPTER 8. SYSTEM CALLS

This chapter describes the PL/M-86 calling sequences for the system calls provided by the Extended I/O System. The list is limited to the calls that can be invoked from application programs. Several system calls provided by the Extended I/O System are reserved for use by system programmers and, consequently, are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

In this chapter, the system calls are listed alphabetically according to the same shorthand notation used throughout this manual. For example, S$CREATE$FILE precedes S$WRITE$MOVE. This 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 Extended I/O System.

CONDITION CODES

The Extended I/O System returns a condition code whenever a system call is invoked. If the call executes without error, the Extended I/O System returns the code E$OK. In contrast, when an error is encountered by the Extended I/O System, the System returns an exceptional condition code. If you are unfamiliar with the purpose of condition codes, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

In each description of a system call, you will find a list of possible condition codes. This list is intended to help you debug your application system. However, it is possible for the Extended I/O System to return condition codes that are not listed. And it is also possible for the Extended I/O System to return the listed codes for reasons that are not listed. To find the numeric value that the Extended I/O System uses to indicate a particular condition code, refer to Appendix C of this manual.

SYSTEM CALL DICTIONARY

The system call dictionary, which is on the next few pages, lists the system calls by function rather than alphabetically. Along with the name of each system call, you will find three pieces of information:

- An extremely brief statement of the purpose of the call.

- A list of the kinds of files on which the call can be used. The abbreviations in this list are as follows:

  PF means physical file.
  SF means stream file.
  NF means named data file.
  ND means named directory file.

- The number of the page on which you can find the detailed description of the system call.

SYSTEM CALLS FOR I/O JOBS

CREATEIOJOB	Creates an I/O job containing one task.		8-4
EXITIOJOB	Sends a message to a previously designated mailbox and deletes the calling task.		8-12

SYSTEM CALLS RELATING TO LOGICAL NAMES

S$CATALOG$CONNECTION	Creates a logical name for a connection by cataloging the connection in the object directory of a specific job.		8-20
S$LOOK$UP$CONNECTION	Searches through an I/O job's local, global, and root object directories to find the connection associated with a logical name.		8-66
S$UNCATALOG$CONNECTION	Deletes a logical name from the object directory of a specific job.		8-99

SYSTEM CALLS FOR CREATING FILES AND CONNECTIONS

S$ATTACH$FILE	Creates a connection to an existing file.	PF SF ND NF	8-15
S$CREATE$DIRECTORY	Creates a new directory file.	ND	8-33
S$CREATE$FILE	Creates a new physical, stream, or named data file. It cannot create a named directory file.	PF SF NF	8-38

SYSTEM CALLS FOR CHANGING ACCESS AND RENAMING

S$CHANGE$ACCESS	Changes the access list for a named file.	ND NF	8-23

S$RENAME$FILE	Changes the path of a named file. It cannot be used for stream or physical files.	ND NF	8-77

SYSTEM CALLS TO MANIPULATE DATA IN FILES

S$CLOSE	Closes an open connection to a named, physical or stream file.	PF SF ND NF	8-30
S$OPEN	Opens a connection to a named, physical, or stream file.	PF SF ND NF	8-69
S$READ$MOVE	Reads a number of bytes from a file to a buffer.	PF SF ND NF	8-73
S$SEEK	Moves the file pointer.	PF ND NF	8-83
S$TRUNCATE$FILE	Removes information from the end of a named data file.	NF	8-96
S$WRITE$MOVE	Writes a collection of bytes from a buffer to a file.	PF SF ND NF	8-101

SYSTEM CALL RELATING DIRECTLY TO DEVICES

S$SPECIAL	Allows your task to perform functions that are peculiar to a specific device.	PF SF	8-87

SYSTEM CALLS FOR OBTAINING STATUS

SGETCON-NECTION$STATUS	Provides status information about file and device connections.	PF SF ND NF	8-53
SGETFILE$STATUS	Allows a task to obtain information about a physical, stream, or named file.	PF SF ND NF	8-57

SYSTEM CALLS TO DELETE FILES AND CONNECTIONS

S$DELETE$CON-NECTION	Deletes a file connection. It cannot delete a device connection.	PF SF ND NF	8-45
S$DELETE$FILE	Deletes a stream, physical, or named file.	PF SF ND NF	8-48

CREATEIOJOB

CREATEIOJOB creates an I/O job containing one task.

```
job = RQ$CREATE$IO$JOB(pool$min, pool$max, except$handler, job$flags,
                       task$priority, start$address, data$seg,
                       stack$ptr, stack$size, task$flags, msg$mbox,
                       except$ptr);
```

INPUT PARAMETERS

pool$min

A WORD containing the minimum allowable size of the new job's pool, in 16-byte paragraphs. For example, a value of 35 indicates thirty-five 16-byte paragraphs. The Extended I/O System also uses this parameter as the initial size of the memory pool for the new job.

You must not assign pool$min a value less than 32. Furthermore, if the base of the stack$ptr parameter is equal to zero, you should ensure that pool$min is no less than 32 + (number of 16-byte paragraphs required to contain the stack). If you set pool$min to a value smaller than these minimums, the Extended I/O System will return an E$PARAM exceptional condition.

The purpose of the pool$min parameter in this system call is identical to the purpose of the pool$min parameter of the CREATE$JOB system call provided by the Nucleus. For information regarding memory pools, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

pool$max

A WORD containing the maximum allowable size of the new job's pool, in 16-byte paragraphs. For example, a value of 40 indicates forty 16-byte paragraphs.

You must set pool$max to a value no less than pool$min, or the Extended I/O System will return an E$PARAM exceptional condition.

The purpose of the pool$max parameter in this system call is identical to the purpose of the pool$max parameter of the CREATE$JOB system call provided by the iRMX 86 Nucleus. For more information about memory pools, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL.

INPUT PARAMETERS (continued)

except$handler A POINTER to a structure of the following form:

```
STRUCTURE(
     exception$handler$ptr      POINTER,
     exception$mode             BYTE)
```

The Extended I/O System 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 wish to designate the system default exception handler, you can do so by setting the base of the exception$handler$ptr to zero. If you set the base to any other value, then the Extended I/O System assumes that the POINTER indicates the first instruction of the exception handler.

Set the exception$mode to tell the Extended I/O System when to pass control to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

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

job$flags A WORD that tells the Nucleus whether to check the validity of objects used as parameters in system calls. If bit 1 (the bit next to the rightmost) is zero, the Nucleus will validate objects. All bits other than bit 1 must be set to zero. This parameter serves precisely the same purpose as the job$flags parameter of the CREATE$JOB system call provided by the Nucleus. Refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for more information.

CREATEIOJOB (continued)

INPUT PARAMETERS (continued)

task$priority — A BYTE which,

- if equal to zero, 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. For more information about the initial job of the Extended I/O System, refer to the chapter of the iRMX CONFIGURATION GUIDE relating to the Extended I/O System.

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

start$address — A POINTER to the first instruction of the code segment for the new job's initial task. This code segment can be, but is not required to be, be an iRMX 86 segment.

data$seg — A WORD which,

- if zero, indicates one of two things. Either the new job's initial task uses no data segment, or it creates one for itself. Tasks can create their own data segments only under special circumstances. To find out more about the circumstances, refer to the iRMX 86 CONFIGURATION GUIDE.

- if not zero, contains the base address of the data segment of the new job's initial task. This data segment can be, but is not required to be, an iRMX 86 segment.

stack$ptr — A POINTER which,

- if the base portion is zero indicates that the Nucleus should allocate a stack for the new job's initial task. The length of the allocated stack is determined by the stack$size parameter of this system call. Be aware that this stack is not an iRMX 86 segment.

INPUT PARAMETERS
stack$ptr (continued)

- if the base portion is not equal to zero, points to the base of the stack for the new job's initial task. Because the Nucleus does not allocate this stack, you must allocate it during the configuration process, or your application code must allocate it while the system is running. Refer to the iRMX 86 CONFIGURATION MANUAL for more information regarding stack allocation.

stack$size — A WORD containing the size, in bytes, of the stack for the new job's initial task. You must set this parameter to 16 or greater, but 200 is the minimum value that you should enter. For information regarding the amount of stack to allocate, refer to the chapter of the iRMX 86 PROGRAMMING TECHNIQUES manual that discusses stack sizes.

If you are allocating the stack during configuration, or if the application code is allocating the stack while the system is running, the value of this parameter will be the precise amount of stack that the system can use. However, if the Nucleus is allocating the stack for you, it might allocate as many as 15 additional bytes in order to make the stack occupy whole 16-byte paragraphs.

task$flags — A WORD in which all bits except the rightmost are zero. Use the rightmost bit (bit 0) to tell the iRMX 86 Operating System whether the new job's initial task uses floating point instructions. A value of 1 indicates the presence of floating point instructions, while a zero indicates the absence of floating point instructions.

msg$mbox — A WORD containing a token for a mailbox. When a task in the I/O job invokes the EXITIOJOB system call, the Extended I/O System will send a message to this mailbox. If you desire no such message, assign msg$mbox a value of zero.

The format of the message is as follows:

```
STRUCTURE(
    termination$code   WORD,
    user$fault$code    WORD,
    job$token          WORD,
    return$data$len    BYTE,
    return$data(*)     BYTE)
```

CREATEIOJOB (continued)

INPUT PARAMETERS
msg$mbox (continued)

where:

termination$code — A WORD that indicates the nature of the termination of the new job. Use the following table to ascertain the cause for termination.

CODE	MEANING
0	Some task within the new job invoked the EXITIOJOB system call. The invoking task did not indicate that any problems caused the termination. The job has not yet been deleted, and some of its tasks might still be ready.
1	The job was deleted because some task invoked the DELETE$JOB system call.
2	Some task within the new job invoked the EXITIOJOB system call and indicated that the job was terminated because some problem occurred. The job has not yet been deleted and some of its tasks might still be ready. The task that invoked the EXITIOJOB system call is known as the terminating task.

user$fault$code — A WORD that contains an encoded reason for the termination of the new job. Whenever the termination$code has a value of 2, this parameter contains an error code (not an exception code) that the terminating task specified when invoking the EXITIOJOB system call. The precise meaning of this code is provided by the terminating task, not by the iRMX 86 Operating System.

job$token — A WORD containing the token of the job that was terminated.

INPUT PARAMETERS (continued)

return$-data$len	A BYTE that specifies the length (in bytes) of the return$data parameter described below. The maximum length is 89 (decimal) bytes.
return$data	A sequence of BYTEs that contain data specified by the terminating task when it invoked the EXITIOJOB system call.

OUTPUT PARAMETERS

job	A WORD in which the Extended I/O System will place a token for the newly created job. This token is valid only if the Extended I/O System returns an E$OK condition code.
except$ptr	A POINTER to a WORD in which the Extended I/O System will place the condition code.

DESCRIPTION

The purpose of this system call is to create a job whose tasks can invoke the system calls provided by the Extended I/O System. Such jobs are called I/O jobs, and they differ from other jobs in three ways:

1. NOTIFICATION OF TERMINATION OF THE NEW JOB

   The CREATEIOJOB system call provides a mechanism for notifying the parent job of the termination of the newly created I/O job. The Extended I/O System implements this mechanism by sending a termination message to a mailbox of your choice whenever a task in the I/O job invokes the EXITIOJOB system call. You specify the mailbox by using the msg$mbox parameter of this system call.

2. CONVENIENT DEFAULTS FOR JOB CREATION PARAMETERS

   Many of the job creation parameters required by the Nucleus's CREATE$JOB system call are not required by the CREATEIOJOB system call. These parameters include:

   directory$size
   param$object
   max$objects
   max$tasks
   max$priority

CREATEIOJOB (continued)

DESCRIPTION (continued)

The Extended I/O System allows you to specify values for some of these parameters during the system initialization process. The precise instructions for defining these values are provided in the iRMX 86 CONFIGURATION GUIDE.

3. CONVENIENT DEFAULTS FOR I/O-RELATED PARAMETERS

The CREATEIOJOB system call provides default values for the following I/O job parameters:

global job
default user
default prefix

The values for these parameters are passed from parent job to child job. For instance, if Job A uses the CREATEIOJOB system call to spawn Job B, then the Extended I/O System will copy the values of the Job A parameters into the Job B parameters. Be aware that if you change the Job A parameters after Job B has been created, the changed values will not be copied into Job B.

You can set the values for these parameters for the "first parent" job during the process of configuring your system. For instructions as to how to set these values, refer to the iRMX 86 CONFIGURATION GUIDE.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The calling task's job is not an I/O job.
E$EXIST	Indicative of any of the following problems: • The token cataloged under the name RQGLOBAL in the directory of the calling task's job does not refer to an existing job. This token is supposed to refer to the global job. • The value assigned to the msg$mbox parameter is not a token for an existing mailbox.
E$MEM	Either the calling task's job or the job being created has insufficient memory.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process:

CONDITION CODES (continued)

ENOTCONFIGURED (continued)

CATALOG$OBJECT (Nucleus)
CREATE$COMPOSITE (Nucleus)
CREATEIOJOB (Extended I/O)
CREATE$JOB (Nucleus)
CREATE$MAILBOX (Nucleus)
CREATE$SEGMENT (Nucleus)
DELETE$JOB (Nucleus)
DELETE$MAILBOX (Nucleus)
DELETE$SEGMENT (Nucleus)
GET$DEFAULT$PREFIX (Basic I/O)
GET$DEFAULT$USER (Basic I/O)
GET$TASK$TOKENS (Nucleus)
LOOKUP$OBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
SEND$CONTROL (Nucleus)
SEND$MESSAGE (Nucleus)
SET$DEFAULT$PREFIX (Basic I/O)
SET$DEFAULT$USER (Basic I/O)
UNCATALOG$OBJECT (Nucleus)

E$NOUSER — The calling task's job does not have a default user, or the object cataloged under the logical name R?USER is not a user object.

E$PARAM — One or more of the following conditions exist:

- The value assigned to the pool$min parameter is too small or is greater than the value assigned to the pool$max parameter.
- The value assigned to the stack$size parameter is less than 16.
- The value assigned to the pool$max parameter is zero.
- The value assigned to the exception$mode parameter is outside the range 0 - 3, inclusive.

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

EXITIOJOB

EXITIOJOB sends a message to a previously designated mailbox and deletes the calling task.

```
CALL RQ$EXIT$IO$JOB(user$fault$code, return$data$ptr, except$ptr);
```

INPUT PARAMETERS

user$fault$code	A WORD containing the encoded reason for termination of the job. If the job is being terminated under normal circumstances, you should enter a value of zero. However, if the job is being terminated because of some problem, you should enter a value that identifies the problem. Whatever value you enter will be passed to the task that invoked the CREATEIOJOB system call.
return$data$ptr	A POINTER to a STRING containing data (provided by the calling task) to be returned to the message mailbox specified in the CREATEIOJOB system call. If you enter a value of zero, no data will be returned. If the string is longer than 89 (decimal) bytes, only the first 89 bytes will be returned.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the Extended I/O System will return the condition code.

DESCRIPTION

The EXITIOJOB system call is designed to complement the CREATEIOJOB system call. Using the EXITIOJOB system call, you can have a task delete itself, and have the Extended I/O System notify the parent job of the deletion.

When a task in an I/O job (a job created by the CREATEIOJOB system call) invokes the EXITIOJOB system call, two things happen:

- The Extended I/O System deletes the task that invoked the EXITIOJOB system call.
- The Extended I/O System sends a termination message to the mailbox specified in the CREATEIOJOB system call.

DESCRIPTION (continued)

Your application code can use this system call to bring about an orderly deletion of an I/O job. To do this, have a task within the I/O job invoke this system call. Then have a task in the parent job receive the message and delete the I/O job.

SPECIAL CIRCUMSTANCES

Under certain circumstances, this system call does not delete the calling task or does not send a termination message.

Calling Task Not Deleted

Because the EXITIOJOB system call generally deletes the calling task, you must be aware of the circumstances under which this deletion will not occur. There are three:

- If the EXITIOJOB system call or the DELETE$TASK system call has not been configured into the system, the calling task will not be deleted.
- If the DELETE$TASK system call (which is called by the Extended I/O System) returns an exception code to the Extended I/O System, your task will not be deleted.
- If the calling task is an interrupt task, it will not be deleted.

If any of these circumstances arise, the Extended I/O System returns control to the calling task and uses an exceptional condition code to indicate the nature of the problem. Under any other circumstances, the calling task will be deleted.

Also, be aware that even if it fails to delete the task, the Extended I/O System will send the termination message if one has been requested.

Message Not Sent

There are three circumstances under which the Extended I/O System will not send a termination message. These are:

- If, during the process of configuring your system, any of the following system calls were not incorporated, the Extended I/O System will delete the calling task but will not send a termination message:

EXITIOJOB (continued)

LOOKUP$OBJECT
RECEIVE$CONTROL
SEND$CONTROL
SEND$MESSAGE

- If the msg$mbox parameter of the CREATEIOJOB was set to zero.
- If the mailbox specified in the msg$mbox parameter of the CREATEIOJOB system call no longer exists.

None of these three circumstances will cause the Extended I/O System to return an exceptional condition.

CONDITION CODES

E$CONTEXT	The task invoking the EXITIOJOB system call is an interrupt task.
ENOTCONFIGURED	When the system was configured, at least one of the following system calls was not incorporated into the system: DELETE$TASK (Nucleus) EXIT$IO$JOB (Extended I/O)

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

```
connection = RQ$S$ATTACH$FILE(path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing the Extended I/O System path for the file to be attached.

OUTPUT PARAMETERS

connection	A WORD in which the Extended I/O System will place a token for the connection to the file.
except$ptr	A POINTER to a WORD where the Extended I/O System will place the condition code.

DESCRIPTION

This system call allows a task to obtain a connection to any named, physical, or stream file.

The Extended I/O System allows any task to attach any file. However, if the file being attached is a named file, the Extended I/O System will compute access rights for the connection. These access rights are based on the file's access list and the id of the default user of the calling task's job. (Refer to the Chapter 5 for more information.) If the file's access list allows no access for the default user, the connection will be created, but it will allow no access.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception can be indicative of any of the following situations: • The device containing the specified file is in the process of being detached. • The calling task's job is not an I/O job.

S$ATTACH$FILE (continued)

CONDITION CODES

E$CONTEXT (continued)

- The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device.

E$DEVFD — When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.

E$FNEXIST — This code is indicative of one of the following circumstances:

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

E$FTYPE — The specified path is attempting to use a data file as a directory.

E$ILLVOL — When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.

E$IO — An I/O error occurred.

E$IOMEM — The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

CONDITION CODES (continued)

E$LIMIT — This can be caused by any of the following conditions:

- While attempting to complete this system call, the Extended I/O System created enough objects to exceed the object limit of the Basic I/O System job. Refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that discusses the Basic I/O System.
- During the process of configuring your application system, someone assigned the Basic I/O System job a maximum priority that is too low. Specifically, the BIOS maximum priority is lower than either the DUIB priority or the DEVINFO priority. Refer to the iRMX 86 CONFIGURATION GUIDE for information regarding the BIOS maximum priority. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEMS for information regarding the DUIB and DEVINFO.
- Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations.
- The calling task's job is not an I/O job.

ELOGNAME$NEXIST — The specified path contains an explicit logical name, but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.

E$MEDIA — The device containing the specified file is not online.

E$MEM — 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.

ENOPREFIX — The specified path contains no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.

S$ATTACH$FILE (continued)

CONDITION CODES (continued)

ENOTCONFIGURED There are two possible conditions that can cause the Extended I/O System to return this code:

- When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that was not configured into your system.

- At least one of the following system calls was left out of the system during the configuration process:

  A$ATTACH$FILE (Basic I/O)
  A$PHYSICAL$ATTACH$DEVICE (Basic I/O)
  A$SPECIAL (Basic I/O)
  CREATE$COMPOSITE (Nucleus)
  CREATE$MAILBOX (Nucleus)
  CREATE$SEGMENT (Nucleus)
  DELETE$COMPOSITE (Nucleus)
  DISABLE$DELETION (Nucleus)
  ENABLE$DELETION (Nucleus)
  GET$DEFAULT$PREFIX (Basic I/O)
  GET$TYPE (Nucleus)
  LOOKUP$OBJECT (Nucleus)
  RECEIVE$CONTROL (Nucleus)
  RECEIVE$MESSAGE (Nucleus)
  S$ATTACH$FILE (Extended I/O)
  SEND$CONTROL (Nucleus)
  SEND$MESSAGE (Nucleus)
  SET$INTERRUPT (Nucleus)
  WAIT$INTERRUPT (Nucleus)

ENOTPREFIX The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.

ENOUSER The calling task's job does not have a default user, or its default user is not a user object.

E$PARAM This code can indicate either of the following conditions:

- The specified path contains a logical name that is either longer than 12 characters or contains invalid characters.

CONDITION CODES

E$PARAM (continued)

- When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.

E$PREFIX$SYNTAX The specified path starts with a colon (:), indicating that it contains a logical name. But the Extended I/O System was unable to find a second colon to terminate the logical name.

S$CATALOG$CONNECTION

The S$CATALOG$CONNECTION system call creates a logical name for a connection by cataloging the connection in the object directory of a specific job. The calling task specifies the connection, the logical name, and the job.

```
CALL RQ$S$CATALOG$CONNECTION(job, connection, log$name$ptr,
                             except$ptr);
```

INPUT PARAMETERS

connection	A WORD containing a token for the connection to be assigned the logical name. If the value of this parameter is zero, the Extended I/O System will obtain the connection by looking up the name in the object directory of the calling task's job.
job	A WORD containing a token for the job in whose object directory the logical name is to be cataloged. If the value of this parameter is zero, the Extended I/O System will catalog the connection in the object directory of the calling task's job.
log$name$ptr	A POINTER to a STRING of 12 or fewer characters that are to become the logical name. These ASCII characters must be between 020h and 07Fh with the exception of colon (:), slash (/), or up arrow (↑). Upper-case letters are considered to be identical to lower-case letters. Any failure to adhere to these specifications will cause the Extended I/O System to return an E$PARAM exception code.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Extended I/O System will place a condition code.

DESCRIPTION

The Extended I/O System converts the contents of the STRING to upper case and catalogs the connection in the object directory of the specified

DESCRIPTION (continued)

job. However, there are two special situations that can change the behavior of this system call:

- If the job's object directory already contains the logical name, the new connection will replace the existing object in the directory. The Extended I/O System considers this to be a normal circumstance and, consequently, does not return an exception code.

- If your task sets the connection parameter to 0, the Extended I/O System will look up the logical name in the object directory of the calling task's job. The system will then copy the logical name and its definition into the object directory of the specified job.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception code can be caused by any of the following circumstances: • The job in which your task is attempting to catalog the connection has an object directory that is zero bytes long. • The UNCATLOG$OBJECT system call was not incorporated into your system during the configuration process.
E$EXIST	The job parameter does not refer to an existing object.
E$LIMIT	The Extended I/O System returns this condition code whenever either of the following circumstances are detected: • The object directory for the specified job is already full. • The calling task's job is not an I/O job.
ELOGNAME$-NEXIST	The Extended I/O System was unable to find the specified logical name in the object directory of the calling task's job.
E$MEM	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.

S$CATALOG$CONNECTION (continued)

CONDITION CODES (continued)

ENOTCONFIGURED	One or more of the following system calls was not incorporated into the System during the process of configuration: CATALOG$OBJECT (Nucleus) CREATE$SEGMENT (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) S$CATALOG$CONNECTION (Extended I/O)
ENOTCONNECTION	This condition code can be indicative of any of the following circumstances: • The connection parameter does not refer to a connection. • The Extended I/O System looked up the specified logical name and found that it did not refer to a connection.
E$PARAM	The STRING specified by the log$name$ptr fails to meet the syntax rules for a logical name. It is either too short (zero characters), too long (more than twelve characters), or it contains invalid characters.
E$TYPE	The job parameter refers to an object that is not a job.

The S$CHANGE$ACCESS system call changes the access list for a named file. This system call can be used for either data or directory files.

```
CALL RQ$S$CHANGE$ACCESS(path$ptr, id, access, except$ptr);
```

INPUT PARAMETERS

path$ptr — A POINTER to a STRING that contains a path to the file whose access is to be changed.

id — A WORD containing the ID of the user whose access to the file is to be changed. This value can differ from the ID of the default user of the calling task's job. The Extended I/O System will add (or remove) this ID to (from) the access list of the file unless (if) the ID is already on the list. Whether the ID is added or deleted also depends upon the value of the access parameter.

access — A BYTE defining the new access rights to be assigned to the specified ID. If the entire BYTE is set to zero, the Extended I/O System will remove the specified ID from the access list of the specified file. If the BYTE is nonzero, the meaning of the various bit settings depend upon whether the file is a data file or a directory file. The following two tables show the correlation between the bit position (bit 0 is rightmost) and the kind of access. If the bit is set to 1, access is to be granted. If the bit is set to 0, access is to be denied.

DATA FILE ACCESS RIGHTS

Bit	Access
0	Delete -- permission to delete the entire file by using the A$DELETE$FILE or S$DELETE$FILE system calls. Also allows changing the name of the file by using the A$RENAME$FILE or S$RENAME$FILE system calls.
1	Read -- permission to read data from the file by using the A$READ or S$READ$MOVE system calls.

S$CHANGE$ACCESS (continued)

INPUT PARAMETERS
access (continued)

Bit	Access
2	Append -- permission to write information only at the end of the file by using the A$WRITE or S$WRITE$MOVE system calls. This does not include permission to write over information already in the file or permission to truncate the file.
3	Update -- permission to write over any information in the file by using the A$WRITE or S$WRITE$MOVE system calls, and permission to truncate the file by using the A$TRUNCATE or S$TRUNCATE$FILE system calls. This does not include permission to add information to the end of the file.
4-7	Reserved. You should ensure that these bits remain set to zero.

DIRECTORY ACCESS RIGHTS

Bits	Access
0	Delete -- permission to delete the directory by using the A$DELETE$FILE or S$DELETE$FILE system calls. Also allows changing the name of the directory by using the A$RENAME$FILE or S$RENAME$FILE system calls.
1	Display -- permission to read information from the directory by using the A$READ or S$READ$MOVE system calls.
2	Add entry -- permission to add files to the directory by using the A$CREATE$FILE, A$CREATE$DIRECTORY, A$RENAME$FILE, S$CREATE$FILE, S$CREATE$DIRECTORY, or S$RENAME$FILE system calls. This does not include permission to change existing entries.

INPUT PARAMTERS
access (continued)

Bit	Access
3	Change entry -- permission to change the access list associated with a file contained in the directory. In other words permission to use the A$CHANGE$ACCESS or S$CHANGE$ACCESS system calls. This does not include permission add new entries.
4-7	Reserved. You should ensure that these bits remain set to zero.

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD where the Extended I/O System is to place the condition code.

DESCRIPTION

The purpose of the S$CHANGE$ACCESS system call is to allow a task to change the access rights associated with named data files or named directory files. This system call can be used on any named files, including those created by the Basic I/O System.

In order for a task to be able to change the access rights associated with a file, the task's job must meet at least one of the following three criteria:

- The job's default user is the owner of the file.
- The owner of the file is a group, and the job's default user is a member of that group.
- The job's default user has change-entry access to the parent directory of the file.

If the job containing the task meets none of these criteria, the task will not be able to change the access associated with the file. (For more information about owners, groups, and default users, refer to Chapter 5 of this manual.)

S$CHANGE$ACCESS (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception can be indicative of any of the following situations: • The device containing the specified file is in the process of being detached. • The calling task's job is not an I/O job. • The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$DEVFD	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$FACCESS	The job containing the calling task meets none of the three prerequisites for using this system call. The job's default user is neither the owner of the file nor a member of the owning group, nor does it have "change entry" access to the parent directory of the file.
E$FNEXIST	This code is indicative of one of the following circumstances: • Either some file in the specified path, or the target file itself, is marked for deletion. • Either some file in the specified path, or the target file itself, does not exist.
E$FTYPE	The specified path is attempting to use a data file as a directory.

CONDITION CODES (continued)

E$ILLVOL	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E$IO	An I/O error occurred.
E$IOMEM	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	This can be caused by any of the following conditions: • While attempting to complete this system call, the Extended I/O System created enough objects to exceed the object limit of the Basic I/O System job. Refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that discusses the Basic I/O System. • During the process of configuring your application system, someone assigned the Basic I/O System job a maximum priority that is too low. Specifically, the BIOS maximum priority is lower than either the DUIB priority or the DEVINFO priority. Refer to the iRMX 86 CONFIGURATION GUIDE for information regarding the BIOS maximum priority. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEMS for information regarding the DUIB and DEVINFO. • Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
ELOGNAME$NEXIST	The specified path contains an explicit logical name, but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.

S$CHANGE$ACCESS (continued)

CONDITION CODES (continued)

E$MEDIA	The device containing the specified file is not on line.
E$MEM	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.
ENOPREFIX	The specified path contains no logical name, so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.
ENOTCONFIGURED	At least one of the following system calls was left out of the system during the configuration process: A$CHANGE$ACCESS (Basic I/O) A$PHYSICAL$ATTACH$DEVICE (Basic I/O) A$SPECIAL (Basic I/O) CREATE$COMPOSITE (Nucleus) CREATE$MAILBOX (Nucleus) CREATE$SEGMENT (Nucleus) DELETE$COMPOSITE (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$DEFAULT$PREFIX (Basic I/O) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) S$CHANGE$ACCESS (Extended I/O) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus)
ENOTPREFIX	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	The calling task's job does not have a default user, or its default user is not a user object.
E$PARAM	This code can indicate either of the following conditions: • The specified path contains a logical name that is either longer than 12 characters or contains invalid characters.

CONDITION CODES

E$PARAM (continued)

- When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.

E$PREFIX$SYNTAX — The specified path starts with a colon (:), indicating that it contains a logical name. But the Extended I/O System was unable to find a second colon to terminate the logical name.

E$SUPPORT — This exception code is indicative of either of the following circumstances:

- Your task is attempting to change access for a file other than a named file.
- Your task is attempting to add another user id to the file's access list, but the list already contains three entries. Your task must delete one entry before it can add another.

S$CLOSE

The S$CLOSE system call closes an open connection to a named, physical, or stream file.

```
CALL RQ$S$CLOSE(connection, except$ptr);
```

INPUT PARAMETER

connection	A WORD containing a token for a file connection that is currently open and was opened by the S$OPEN system call.

OUTPUT PARAMETER

except$ptr	A POINTER to the WORD in which the Extended I/O System will place the condition code.

DESCRIPTION

The S$CLOSE system call closes a connection that has been opened by the S$OPEN system call. It performs the following steps:

1. It waits until all currently running I/O operations for the file are completed.
2. It ensures that any information in a partially filled output buffer is written to the file.
3. It releases any buffers associated with the file.
4. It closes the connection to the file, deleting neither the file nor the connection.

Access Control

The Extended I/O System performs no access checking before closing the connection.

DESCRIPTION (continued)

Basic I/O System Incompatibility

The S$CLOSE system call cannot be used to close connections that were opened by the A$OPEN system call. If your task attempts to do this, the Extended I/O System will return an E$CONTEXT exception code. Refer to Appendix E for more information about compatibility between the Extended and Basic I/O Systems.

CONDITION CODES

E$OK	No exceptional conditions.
E$CANNOT$CLOSE	This system call forced the Extended I/O System to write information remaining in the buffers to the output device. While attempting to write this information, the Extended I/O System encountered an error.
E$CONTEXT	Either the connection is not open, or it was opened by A$OPEN rather than S$OPEN.
E$FLUSHING	The Extended I/O System terminated the closing operation because the device containing the file is being detached.
E$IO	The Extended I/O System encountered an I/O error.
E$LIMIT	This code can be indicative of any of the following circumstances: • The calling task's job is not an I/O job. • The calling task's job is currently involved in more than 255 (decimal) I/O operations.
E$MEM	The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.
ENOTCON-FIGURED	At least one of the following system calls was not incorporated into the system during the configuration process:

S$CLOSE (continued)

CONDITION CODES

ENOTCONFIGURED (continued)

A$CLOSE (Basic I/O)
A$WRITE (Basic I/O)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)
ENABLE$DELETION (Nucleus)
GET$TYPE (Nucleus)
LOOKUP$OBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
S$CLOSE (Extended I/O)
SEND$CONTROL (Nucleus)
SEND$MESSAGE (Nucleus)

ENOTCONNECTION	The connection parameter refers to an object that is not a connection.
E$SPACE	In order to successfully execute this system call, the Extended I/O System had to write to the device containing the file. While this writing operation was only partially complete, the device became full, preventing the Extended I/O System from completing the writing operation.
E$SUPPORT	The connection parameter refers to a connection that was not created within the calling task's job.

The S$CREATE$DIRECTORY system call creates a new directory file.

```
connection = RQ$S$CREATE$DIRECTORY(path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing the path that specifies the location of the new directory in the named file structure.

OUTPUT PARAMETERS

connection	A WORD in which the Extended I/O System will place a connection for the new directory.
except$ptr	A POINTER to a WORD where the Extended I/O System will place a condition code.

DESCRIPTION

A task invokes this system call to create a new named-file directory. After creation, the new directory will contain no entries. The new directory is in all ways compatible with directories created by the Basic I/O System. The difference between this system call and the A$CREATE$DIRECTORY system call of the Basic I/O System is that this call is synchronous, requires less explicit parameters, and uses an Extended I/O path to specify the file.

Positioning the Directory

The calling task must use the path$ptr parameter to specify the location of the new directory within the named file structure. The location indicated by the path must not be occupied. In other words, this system call can be used only to obtain connections to new, rather than existing, directories.

S$CREATE$DIRECTORY (continued)

DESCRIPTION (continued)

Access Control

The current default user for the calling task's job must have add-entry access to the parent of the new directory. If the creation is successful, the job's default user becomes the owner of the file.

One by-product of this system call is that the Extended I/O System will automatically add a new entry to the parent directory. This new entry, which represents the newly created directory, provides the owner of the new directory with full access (the ability to delete, display, change, and add entries) to the new directory.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception can be indicative of any of the following situations: • The device containing the specified file is in the process of being detached. • The calling task's job is not an I/O job. • The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$DEVFD	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$FEXIST	The file already exists.
E$FNEXIST	This code is indicative of one of the following circumstances: • Either some file in the specified path, or the target file itself, is marked for deletion. • Some file in the specified path does not exist.

CONDITION CODES (continued)

E$FTYPE	The specified path is attempting to use a data file as a directory.
E$ILLVOL	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E$IO	An I/O error occurred.
E$IOMEM	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	This can be caused by any of the following conditions: • While attempting to complete this system call, the Extended I/O System created enough objects to exceed the object limit of the Basic I/O System job. Refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that discusses the Basic I/O System. • During the process of configuring your application system, someone assigned the Basic I/O System job a maximum priority that is too low. Specifically, the BIOS maximum priority is lower than either the DUIB priority or the DEVINFO priority. Refer to the iRMX 86 CONFIGURATION GUIDE for information regarding the BIOS maximum priority. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEMS for information regarding the DUIB and DEVINFO. • Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations. • The calling task's job is not an I/O job.

S$CREATE$DIRECTORY (continued)

CONDITION CODES (continued)

ELOGNAME- $NEXIST	The specified path contains an explicit logical name, but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.
E$MEDIA	The device containing the specified file is not online.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
ENOPREFIX	The specified path contains no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.
ENOTCON-- FIGURED	At least one of the following system calls was left out of the system during the configuration process: A$CREATE$DIRECTORY (Basic I/O) A$PHYSICAL$ATTACH$DEVICE (Basic I/O) A$SPECIAL (Basic I/O) CREATE$COMPOSITE (Nucleus) CREATE$MAILBOX (Nucleus) CREATE$SEGMENT (Nucleus) GET$DEFAULT$PREFIX (Basic I/O) DELETE$COMPOSITE (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) S$CREATE$DIRECTORY (Extended I/O) SET$INTERRUPT (Nucleus) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus) WAIT$INTERRUPT (Nucleus)
ENOTPREFIX	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	The calling task's job does not have a default user, or its default user is not a user object.
E$PARAM	This code can indicate either of the following conditions:

CONDITION CODES
E$PARAM (continued)

- The specified path contains a logical name that is either longer than 12 characters or contains invalid characters.
- When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.

E$PREFIX$SYNTAX — The specified path starts with a colon (:), indicating that it contains a logical name. But the Extended I/O System was unable to find a second colon to terminate the logical name.

E$SPACE — This code can indicate any of the following circumstances:

- There is no more space on the volume.
- The Extended I/O System has run out of fnodes on the volume. Refer to the iRMX 86 CONFIGURATION GUIDE to see how to put more fnodes on a volume.

S$CREATE$FILE

The S$CREATE$FILE system call creates a new physical, stream, or named data file. It cannot create a named directory file.

```
connection = RQ$S$CREATE$FILE(path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING that specifies the path of the file to be created. The format of this path depends on the kind of file being created. Refer to Chapter 5 for a discussion of named file paths, Chapter 6 for physical file paths, and Chapter 7 for stream file paths.

OUTPUT PARAMETERS

connection	A WORD in which the Extended I/O System will place a connection to the newly created file.
except$ptr	A POINTER to a WORD where the Extended I/O System will place a condition code.

DESCRIPTION

A task invokes this system call to create a physical, stream, or named data file, or to attach an existing file. This system call cannot be used to create or attach a directory. (The Extended I/O System provides the S$CREATE$DIRECTORY system call for that purpose.) The file created by this system call is in all ways compatible with the files created by the Basic I/O System. The primary differences between this system call and the A$CREATE$FILE system call of the Basic I/O System is that this system call is synchronous and requires fewer explicit parameters.

Creating Data Files that Already Exist

If the specified file already exists, the Extended I/O System will attempt to remove all information from the file and return a connection to the empty file. This attempt will be successful only if both of the following conditions exist at the time the system call is invoked:

DESCRIPTION (continued)

- All connections to the file that are currently open allow sharing with writers.

- If the file is a named file, the default user of the calling task's job has update access to the existing file.

If you wish to prevent this loss of information from happening, use the S$ATTACH$FILE system call to find out if the file exists before using the S$CREATE$FILE system call.

Specifying the Kind of File to Be Created

The path$ptr parameter does more than simply provide the path of the file being created. It also tells the Extended I/O System what kind of file (stream, physical, or named data) to create. The correlation between file paths and the kinds of files is discussed in detail in Chapters 5, 6, and 7.

Special Considerations for Named Files

There are three special considerations that relate to named files:

- Positioning a Named File

  Your task must tell the Extended I/O System which directory is to be the parent of the new named file.

- Access Control

  Several aspects of access control relate to the creation of named files. They are:

  - Ability to Create the File

    In order to create a named file, the default user for the calling task's job must have add-entry access for the parent directory.

  - Ownership

    The default user for the calling task's job becomes the owner of the new file.

S$CREATE$FILE (continued)

DESCRIPTION
Special Considerations for Named Files (continued)

- Initial Access Rights

  The new file is created with full access for the owner. In other words, the owner can delete, read, append, and update the file. If the owner is a group, the members of the group can also access the file. However, if the owner is not a group, no other users can access the new file until the owner explicitly changes the file's access list.

- Temporary Files

  If your task invokes this system call with the path of an existing directory file, the Extended I/O System will create a temporary named data file on the device that contains the directory file. This temporary file differs from other named data files in two ways. First, the file is automatically marked for deletion, so that Extended I/O System will delete the file as soon as your application code deletes all connections to the file. Second, the file is created without a path, so it can be accessed only through a connection.

  There are two access considerations that pertain to temporary files. First, any task can create a temporary file by referring to any directory. This is true because the temporary files are not listed as ordinary entries in the directory, so no add-entry access is required.

  The second access consideration is that the owner of the temporary file (the default user of the calling task's job) has full access to the file.

Device Considerations

Every file, regardless of kind, has an associated device. Even stream files, which have no physical devices, use the device connection to the stream file pseudo-device. Before any file can be created, its associated device must be attached to the system.

There are two means of attaching devices to the system. One method is to specify the attachment during configuration. This process is described in the iRMX 86 CONFIGURATION GUIDE.

The second method is to attach a device while the system is running. This method requires the use of the LOGICAL$ATTACH$DEVICE system call. Because the process of attaching devices is generally performed by system programs, the LOGICAL$ATTACH$DEVICE system call is described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

DESCRIPTION (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception can be indicative of any of the following situations: • The device containing the specified file is in the process of being detached. • The calling task's job is not an I/O job. • The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$DEVFD	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$FACCESS	This code can be indicative of any of the following circumstances: • The default user of the calling task's job does not have add-entry access to the parent directory. • The default user of the calling task's job does not have update access for an existing file.
E$FNEXIST	This code is indicative of one of the following circumstances: • Either some file in the specified path, or the target file itself, is marked for deletion. • Some file in the specified path does not exist.

S$CREATE$FILE (continued)

CONDITION CODES (continued)

E$FTYPE	The specified path is attempting to use a data file as a directory.
E$ILLVOL	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E$IO	An I/O error occurred.
E$IOMEM	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	This can be caused by any of the following conditions:

- While attempting to complete this system call, the Extended I/O System created enough objects to exceed the object limit of the Basic I/O System job. Refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that discusses the Basic I/O System.

- During the process of configuring your application system, someone assigned the Basic I/O System job a maximum priority that is too low. Specifically, the BIOS maximum priority is lower than either the DUIB priority or the DEVINFO priority. Refer to the iRMX 86 CONFIGURATION GUIDE for information regarding the BIOS maximum priority. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEMS for information regarding the DUIB and DEVINFO.

- Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations.

- The calling task's job is not an I/O job.

CONDITION CODES (continued)

ELOGNAME$NEXIST	The specified path contains an explicit logical name but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.
E$MEDIA	The device containing the specified file is not on line.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
ENOPREFIX	The specified path contains no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.
ENOTCONFIGURED	At least one of the following system calls was left out of the system during the configuration process: A$CREATE$FILE (Basic I/O) A$PHYSICAL$ATTACH$DEVICE (Basic I/O) A$SPECIAL (Basic I/O) CREATE$COMPOSITE (Nucleus) CREATE$MAILBOX (Nucleus) CREATE$SEGMENT (Nucleus) DELETE$COMPOSITE (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$DEFAULT$PREFIX (Basic I/O) GET$TYPE (Nucleus) LOOKUPOBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) S$CREATE$FILE (Extended I/O) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus) SET$INTERRUPT (Nucleus) WAIT$INTERRUPT (Nucleus)
ENOTPREFIX	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	The calling task's job does not have a default user, or its default user is not a user object.

S$CREATE$FILE (continued)

CONDITION CODES (continued)

E$PARAM	This code can indicate either of the following conditions: • The specified path contains a logical name that is either longer than 12 characters or contains invalid characters. • When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.
E$PREFIX$SYNTAX	The specified path starts with a colon (:), indicating that it contains a logical name. But the Extended I/O System was unable to find a second colon to terminate the logical name.
E$SHARE	Your task is attempting to create a file that already exists. Censequently, the Extended I/O System must truncate the file to zero length. However, the Extended I/O System cannot do this because, when the file was initially created, the owner specified that it could not be shared with writers.
E$SPACE	This code can indicate any of the following circumstances: • There is no more space on the volume. • The Extended I/O System has run out of fnodes on the volume. Refer to the format command in the iRMX 86 HUMAN INTERFACE REFERENCE MANUAL.
E$SUPPORT	Your task is attempting to create an existing file. Consequently, the Extended I/O System must change the file's access list. However, the access list is already full.

The S$DELETE$CONNECTION system call deletes a file connection. It cannot delete a device connection.

```
CALL RQ$S$DELETE$CONNECTION(connection, except$ptr);
```

INPUT PARAMETER

connection	A WORD containing a token for the file connection to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Extended I/O System is to place the condition code.

DESCRIPTION

This system call deletes a file connection , but it cannot delete a device connection. If the connection is open, the S$DELETE$CONNECTION system call will automatically close it before deleting it.

Conditions Under Which the File is Also Deleted

The Extended I/O System will delete the file that is associated with the connection only when both of the following conditions are met:

- The connection being deleted is the only connection to the file.
- The file is marked for deletion. A file is marked for deletion only if both of the following events have already occurred:
  - Some task attempted to delete the file by invoking the S$DELETE$FILE system call or the A$DELETE$FILE system call. (The A$DELETE$FILE system call is provided by the Basic I/O System.)
  - The iRMX 86 Operating System postponed the deletion because one or more connections still referred to the file. This implies that some other tasks were still using the file.

S$DELETE$CONNECTION (continued)

DESCRIPTION (continued)

If both of these conditions are met, the Extended I/O System will delete the file.

Access Control

The Extended I/O System does not check access before deleting a connection.

Compatibility with Basic I/O System

The S$DELETE$CONNECTION system call can be used with connections that were created by the Basic I/O System as long as the connections meet the requirements discussed in Appendix E. Failure to adhere to the restrictions of Appendix E will cause the Extended I/O System to return an E$CONTEXT condition code.

CONDITION CODES

E$OK	No exceptional conditions.
E$CANNOT$CLOSE	This system call forced the Extended I/O System to write information remaining in the buffers to the output device. While attempting to write this information, the Extended I/O System encountered an error.
E$FLUSHING	The Extended I/O System had to close the connection before it could delete it. However, the Extended I/O System was unable to close the connection because the device containing the file was being detached.
E$IO	The Extended I/O System encountered an I/O error.
E$LIMIT	This code can be indicative of any of the following circumstances: • The associated job or the job's default user object is currently involved with more than 255 (decimal) I/O operations. • The calling task's job is not an I/O job.

CONDITION CODES (continued)

E$MEM	The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: A$DELETE$CONNECTION (Basic I/O) A$WRITE (Basic I/O) CREATE$MAILBOX (Nucleus) CREATE$SEGMENT (Nucleus) DELETE$COMPOSITE (Nucleus) DELETE$MAILBOX (Nucleus) DELETE$REGION (Nucleus) DELETE$SEGMENT (Nucleus) DELETE$TASK (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) S$DELETE$CONNECTION (Extended I/O) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus)
ENOTCONNECTION	The connection parameter is not a connection.
E$SPACE	In order to successfully execute this system call, the Extended I/O System had to write to the device containing the file. While this writing operation was only partially complete, the device became full, preventing the Extended I/O System from completing the writing operation.
E$SUPPORT	Your task is attempting to delete a connection that was created by a task in a different job.

S$DELETE$FILE

A task can invoke this system call to request that the Extended I/O System delete a stream, named data, or named directory file. This system call cannot delete a physical file.

```
CALL RQ$S$DELETE$FILE(path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING that specifies the path for the file to be deleted. The form of the path depends upon the kind of file. Refer to Chapters 5 and 7 for path syntax.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Extended I/O System will place a condition code.

DESCRIPTION

A task can use this system call whenever the task needs to delete a stream, named data, or named directory file. This system call will mark the specified file for deletion, but the Extended I/O System will actually postpone deletion until the following conditions are met:

- For stream and named data files, there is only one condition. The deletion will occur as soon as there are no longer any connections referring to the file. Your tasks can use the S$DELETE$CONNECTION system call to delete connections.
- For named directories there are two conditions. The directory must be empty, and there can be no connections referring to the directory. The Extended I/O System will delete marked directories as soon as both of these conditions are met.

Compatibility with Basic I/O System

This system call can delete files created by the Basic I/O System as well as those created by the Extended I/O System. Refer to Appendix E for a more general discussion of compatibility between the Extended and Basic I/O Systems.

DESCRIPTION (continued)

Access Considerations

If the task is attempting to delete a named data or directory file, the task must satisfy one access requirement before the Extended I/O System can mark the file for deletion. The default user of the task's job must have deletion access to the file.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception can be indicative of any of the following situations: • The device containing the specified file is in the process of being detached. • The calling task's job is not an I/O job. • The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device. • Your task is attempting to delete a directory that is either a root directory or is not empty.
E$DEVFD	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$FACCESS	The default user of the calling task's job does not have delete access for specified file.
E$FNEXIST	This code is indicative of one of the following circumstances: • Some file in the specified path is marked for deletion. • Either some file in the specified path, or the target file itself, does not exist.

S$DELETE$FILE (continued)

CONDITION CODES (continued)

E$FTYPE	The specified path is attempting to use a data file as a directory.
E$ILLVOL	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E$IO	An I/O error occurred.
E$IOMEM	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	This can be caused by any of the following conditions:

- While attempting to complete this system call, the Extended I/O System created enough objects to exceed the object limit of the Basic I/O System job. Refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that discusses the Basic I/O System.

- During the process of configuring your application system, someone assigned the Basic I/O System job a maximum priority that is too low. Specifically, the BIOS maximum priority is lower than either the DUIB priority or the DEVINFO priority. Refer to the iRMX 86 CONFIGURATION GUIDE for information regarding the BIOS maximum priority. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEMS for information regarding the DUIB and DEVINFO.

- Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations.

- The calling task's job is not an I/O job.

CONDITION CODES (continued)

ELOGNAME$-NEXIST	The specified path contains an explicit logical name, but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.
E$MEDIA	The device containing the specified file is not on line.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
ENOPREFIX	The specified path contains no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.
ENOTCONFIGURED	At least one of the following system calls was left out of the system during the configuration process: A$DELETE$FILE (Basic I/O) A$PHYSICAL$ATTACH$DEVICE (Basic I/O) A$SPECIAL (Basic I/O) CREATE$COMPOSITE (Nucleus) CREATE$MAILBOX (Nucleus) CREATE$SEGMENT (Nucleus) DELETE$COMPOSITE (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$DEFAULT$PREFIX (Basic I/O) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) S$DELETE$FILE (Extended I/O) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus) SET$INTERRUPT (Nucleus) WAIT$INTERRUPT (Nucleus)
ENOTPREFIX	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	The calling task's job does not have a default user, or its default user is not a user object.

S$DELETE$FILE (continued)

CONDITION CODES (continued)

E$PARAM	This code can indicate either of the following conditions: • The specified path contains a logical name that is either longer than 12 characters or contains invalid characters. • When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. The Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.
E$PREFIX$SYNTAX	The specified path starts with a colon (:), indicating that it contains a logical name. But the Extended I/O System was unable to find a second colon to terminate the logical name.
E$SUPPORT	Your task is attempting to delete a physical file.

The SGETCONNECTION$STATUS system call provides status information about file and device connections.

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

INPUT PARAMETER

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

OUTPUT PARAMETERS

except$ptr — A POINTER to a WORD in which the Extended I/O System will place the condition code.

info$ptr — A POINTER to a structure in which the Extended I/O System will place the status information. You must provide the memory for this structure by requesting an iRMX 86 segment. The structure must have the following form:

```
connection$info      STRUCTURE(
   file$driver          BYTE,
   flags                BYTE,
   open$mode            BYTE,
   share$mode           BYTE,
   low$file$pointer     WORD,
   high$file$pointer    WORD,
   access               BYTE,
   num$buf              BYTE,
   buf$size             WORD,
   seek                 BOOLEAN)
```

where:

file$driver — Identifies the kind of file associated with the connection.

1 means physical file
2 means stream file
3 means named file

SGETCONNECTION$STATUS (continued)

OUTPUT PARAMETERS
info$ptr (continued)

flags	Uses individual bits to indicate what kind of connection this is. If Bit 1 (next to rightmost) is set to one, the connection is capable of being opened. If Bit 2 is set to one, the connection is a device connection.
open$mode	Indicates the purpose for which the connection was opened. This applies only to file connections. 0 means closed. 1 means open for reading only. 2 means open for writing only. 3 means open for both reading and writing.
share$mode	Indicates who may share the connection. Applies to both device and file connections. 0 means cannot be shared. 1 means share with readers only. 2 means share with writers only. 3 means share with anybody.
file$pointer	These two words form a 32-bit POINTER to the information in the file. This POINTER shows where the next I/O operation will be performed.
access	This byte specifies the access rights that were computed when the connection was opened. This information applies only to connections for named files, and the interpretation of the information depends upon whether the file is a data file or a directory. Access is represented as a bit mask. In the following tables Bit 0 is the rightmost, and access is granted if a bit is set to one.

OUTPUT PARAMETERS
info$ptr (continued)

Bit	Data File	Directory
0	Delete	Delete
1	Read	Display
2	Append	Add Entry
3	Update	Change Entry
4-7	Reserved	Reserved

num$buf — The number of buffers to be used with this connection. This applies only to file connections.

buf$size — Contains the size, in bytes, of each buffer.

seek — Tells whether or not the SEEK function can be used with this connection. Zero means no, and 0FFh means yes.

DESCRIPTION

The SGETCONNECTION$STATUS system call allows a task to obtain status information about file connections and device connections that were created by either the Basic I/O System or the Extended I/O System. The nature of the returned information depends upon whether the connection is for a file or a device. Some of the information is also dependent upon the kind of file associated with the connection. So be aware that by using this system call, you might lock your application into a specific kind of file.

Access Control

The Extended I/O System does not check access before returning status information.

Incompatibility with Basic I/O System

Although you can use this system call with connections created by the Basic I/O System, you must adhere to the restrictions described in Appendix E. Failure to adhere to these restrictions will cause the Extended I/O System to return an E$CONTEXT condition code.

SGETCONNECTION$STATUS (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	This code can be indicative of any of the following circumstances: • Either the calling task's job, or the job's default user object, is currently involved in more than 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: AGETFILE$STATUS (Basic I/O) CREATE$MAILBOX (Nucleus) CREATE$SEGMENT (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus) S$GET$CONNECTION$STATUS (Extended I/O)
ENOTCONNECTION	The connection parameter is not a connection.

The SGETFILE$STATUS system call allows a task to obtain information about a physical, stream, or named file.

```
CALL RQ$S$GET$FILE$STATUS(path$ptr, info$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr — A POINTER to a STRING that contains the path for the file. The format of this path varies from one kind of file to another. Refer to Chapters 5, 6, or 7 for path syntax.

OUTPUT PARAMETERS

except$ptr — A POINTER to a WORD in which the Extended I/O System will place a condition code.

info$ptr — A POINTER to a structure in which the Extended I/O System will return the status information. You must allocate this memory as an iRMX 86 segment. The structure has the following form:

```
file$info               STRUCTURE(
   device$share             WORD,
   file$conn                WORD,
   file$reader              WORD,
   file$writer              WORD,
   share                    BYTE,
   extended                 BOOLEAN,
   dev$name(14)             BYTE,
   file$drivers             WORD,
   functs                   WORD,
   dev$gran                 WORD,
   low$dev$size             WORD,
   high$dev$size            WORD,
   dev$con                  WORD,
   file$id                  WORD,
   file$type                BYTE,
   file$gran                BYTE,
   owner$id                 WORD,
   low$creation$time        WORD,
   high$creation$time       WORD,
   low$access$time          WORD,
   high$access$time         WORD,
```

SGETFILE$STATUS (continued)

OUTPUT PARAMETERS
info$ptr (continued)

```
low$modify$time          WORD,
high$modify$time         WORD,
low$file$size            WORD,
high$file$size           WORD,
low$file$blocks          WORD,
high$file$blocks         WORD,
vol$name(6)              BYTE,
vol$gran                 WORD,
low$vol$size             WORD,
high$vol$size            WORD,
accessor$count           WORD,
owner$access             BYTE)
```

where:

device$share	indicates whether or not the device can be shared. Currently, this word is always set to 1, indicating that all devices are sharable.
file$conn	is the number of connections to the file.
file$reader	is the number of connections currently opened for reading this file.
file$writer	is the number of connections currently open for writing to the file.
share	is the current shared state of the file. The Extended I/O System always allows the file to be shared by both readers and writers.
extended	tells whether this structure contains any information beyond the dev$conn field. FFh means yes and 0 means no.

OUTPUT PARAMETERS
info$ptr (continued)

dev$name — is the name of the physical device associated with this file. The name is padded with ASCII blanks. For information about device names, refer to the chapter in the iRMX 86 CONFIGURATION GUIDE that refers to the Basic I/O System.

file$drivers — is a bit mask that tells what kinds of files can reside on this device. To interpret this mask, use the following table. If the bit is set to 1, the device can contain the corresponding kind of file. Bit 0 is the rightmost.

Bit	Kind of File
0	Physical file
1	Stream file
2	Reserved
3	Named file
4-15	Reserved

functs — is a bit map that tells what functions can be used with the device associated with the specified file. Bit 0 is the rightmost. If a bit is set to 1, the corresponding function can be used on the device.

Bit	Function
0	Read
1	Write
2	Seek
3	Special
4	Attach Device
5	Detach Device
6	Open
7	Close
8-15	Reserved

dev$gran — is the granularity, in bytes, of the device on which the file resides.

SGETFILE$STATUS (continued)

OUTPUT PARAMETERS

info$ptr (continued)

lowdevsize and highdevsize	combine to form a 32-bit integer that contains the storage capacity, in bytes, for the device on which the file resides.
dev$conn	The number of connections (both file and device connections) to the device on which the file resides.
file$id	is a number that distinguishes this file from all other files on the same device.
file$gran	specifies the granularity of the file in multiples of the vol$gran. For example, if file$gran is 2, and vol$gran is 256, then the file's granularity is 2 * 256.
owner$id	is the id of the user object that was presented to the System when the file was created.
low$creation$time and high$creation$time	combine to form a 32-bit integer that contains the time of the creation of the file. The maintenance of this field is an option that you can select during the configuration of the Basic I/O System.
low$access$time and high$access$time	combine to form a 32-bit integer containing the time at which this file was last accessed. Maintenance of this information is an option that you can select during the configuration of the Basic I/O System.
low$modify$time and high$modify$time	combine to form a 32-bit integer containing the time at which this file was last modified. Maintenance of this information is an option that you can select during the configuration of the Basic I/O System.

OUTPUT PARAMETERS
info$ptr (continued)

low$file$size and high$file$size	combine to form a 32-bit integer containing the current size of the file.
low$file$blocks and high$file$blocks	combine to form a 32-bit integer containing the number of volume blocks allocated to this file. A volume block is a contiguous chunk of storage that can contain vol$gran bytes of information.
vol$name	is the six-character ASCII name of the volume containing this file.
vol$gran	is the size, in bytes, of each volume block for the volume containing this file.
lowvolsize and highvolsize	combine to form a 32-bit integer that contains the storage capacity, in bytes, of the volume on which this file is stored.
accessor$count	contains the number of user id's (not counting the owner of the file) that have access to this file. This information is meaningful only for named files.
owner$access	contains the access rights to this file that are currently held by the owner. This information is meaningful only for named files. The access rights are encoded in a bit mask that you can interpret by using the following table. Remember that Bit 0 is the rightmost bit, and that access is granted if the corresponding bit is set to 1.

SGETFILE$STATUS (continued)

OUTPUT PARAMETERS
info$ptr (continued)

Bit	Data File	Directory
0	Delete	Delete
1	Read	Display
2	Append	Add Entry
3	Update	Change Entry
4-7	reserved	reserved

DESCRIPTION

This system call provides the calling task with information about the status of a file. All fields through the dev$conn field are always returned. Fields following the dev$conn field are returned only when the contents of the extended field are FFh.

Incompatibility with Basic I/O System

This system call can be used with any file, including those created by the Basic I/O System. However, because of the asynchronous nature of some of the system calls provided by the Basic I/O System, there is some chance that the returned information may be inaccurate. For instance, if your application code invokes the SGETFILE$STATUS system call while the Basic I/O System is processing an A$WRITE for the same file, the values returned in the file size fields might be incorrect. Refer to Appendix E for a more general discussion of compatibility between the Extended and Basic I/O Systems.

Access Control

The Extended I/O System does not check access before returning file status information.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception can be indicative of any of the following situations:

CONDITION CODES
E$CONTEXT (continued)

- The device containing the specified file is in the process of being detached.
- The calling task's job is not an I/O job.
- The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device.

E$DEVFD — When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.

E$FNEXIST — This code is indicative of one of the following circumstances:

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

E$FTYPE — The specified path is attempting to use a data file as a directory.

E$ILLVOL — When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.

E$IO — An I/O error occurred.

E$IOMEM — The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

SGETFILE$STATUS (continued)

CONDITION CODES (continued)

E$LIMIT — This can be caused by any of the following conditions:

- While attempting to complete this system call, the Extended I/O System created enough objects to exceed the object limit of the Basic I/O System job. Refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that discusses the Basic I/O System.
- During the process of configuring your application system, someone assigned the Basic I/O System job a maximum priority that is too low. Specifically, the BIOS maximum priority is lower than either the DUIB priority or the DEVINFO priority. Refer to the iRMX 86 CONFIGURATION GUIDE for information regarding the BIOS maximum priority. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEMS for information regarding the DUIB and DEVINFO.
- Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations.
- The calling task's job is not an I/O job.

ELOGNAME$NEXIST — The specified path contains an explicit logical name, but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.

E$MEDIA — The device containing the specified file is not online.

E$MEM — 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.

ENOPREFIX — The specified path contains no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.

ENOTCONFIGURED — At least one of the following system calls was left out of the system during the configuration process:

CONDITION CODES
ENOTCONFIGURED (continued)

A$ATTACH$FILE (Basic I/O)
AGETFILE$STATUS (Basic I/O)
A$PHYSICAL$ATTACH$DEVICE (Basic I/O)
A$SPECIAL (Basic I/O)
CREATE$COMPOSITE (Nucleus)
CREATE$MAILBOX (Nucleus)
CREATE$SEGMENT (Nucleus)
DELETE$COMPOSITE (Nucleus)
DISABLE$DELETION (Nucleus)
ENABLE$DELETION (Nucleus)
GET$DEFAULT$PREFIX (Basic I/O)
GET$TYPE (Nucleus)
LOOKUP$OBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
SEND$CONTROL (Nucleus)
SEND$MESSAGE (Nucleus)
SET$INTERRUPT (Nucleus)
SGETFILE$STATUS (Extended I/O)
WAIT$INTERRUPT (Nucleus)

ENOTPREFIX — The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.

ENOUSER — The calling task's job does not have a default user, or its default user is not a user object.

E$PARAM — This code can indicate either of the following conditions:

- The specified path contains a logical name that is either longer than 12 characters or contains invalid characters.
- When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.

E$PREFIX$SYNTAX — The specified path starts with a colon (:), indicating that it contains a logical name. But the Extended I/O System was unable to find a second colon to terminate the logical name.

S$LOOK$UP$CONNECTION

The S$LOOK$UP$CONNECTION system call accepts a logical name from the calling task and returns the connection associated with the logical name.

```
connection = RQ$S$LOOK$UP$CONNECTION(log$name$ptr, except$ptr);
```

INPUT PARAMETER

log$name$ptr	A POINTER to a STRING containing the logical name under which the connection is cataloged. This string can contain as many as 12 ASCII characters that lie between 020h and 07Fh. However, the Extended I/O System will convert all lower-case letters to upper case before looking up the connection.

OUTPUT PARAMETERS

connection	A WORD in which the Extended I/O System will place the token for the connection associated with the logical name.
except$ptr	A POINTER to a WORD in which the Extended I/O System will place a condition code.

DESCRIPTION

This system call is performed in two steps. The first step is translation, and the second step is looking up the connection.

Translation

Before looking up the connection, the Extended I/O System converts any lower-case letters in the logical name to upper case.

DESCRIPTION (continued)

Looking up the Connection

After the translation, the Extended I/O System tries to find the logical name in the object directories of as many as three jobs. First the Extended I/O System checks the object directory of the local job, then the global job, and finally the root job. After the first successful lookup, the Extended I/O System ceases checking any other object directories, and returns the token for the connection.

Local Job. A task's local job is the job that owns the task.

Global Job. A task's global job can be found by looking up the logical name RQGLOBAL in the object directory of the local job. If there is no such entry, or if the associated object is not a job, then the task has no global job. If there is a job cataloged under RQGLOBAL, it is task's global job.

Whenever a job is created by the CREATEIOJOB system call, the new job inherits the parent's global job. This happens because the CREATEIOJOB system call copies the RQGLOBAL entry from the parent to the offspring.

Currently, the iRMX 86 Human Interface requrires that the RQGLOBAL entry of every I/O job be set to a specific value. Consequently, if your system uses the Human Interface, your tasks must not modify the RQGLOBAL entry.

Root Job. The iRMX 86 System has one root job. This job is discussed in the iRMX 86 CONFIGURATION GUIDE. If you wish to catalog objects in the object directory of the root job, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual, particularly the chapter that deals with communication between tasks and jobs.

Compatibility with Nucleus

Your tasks can invoke this system call to look up logical names created by the CATALOG$OBJECT system call provided by the Nucleus. However, the Nucleus system call does not translate from lower to upper case and does not verify that the ASCII codes for the characters lie between 20h and 7Fh.. So if you desire compatibility, restrict your alphabetic characters when you use the CATALOG$OBJECT system call.

S$LOOK$UP$CONNECTION (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The calling task's job is not an I/O job.
E$LIMIT	The calling task's job is not an I/O job.
ELOGNAME$NEXIST	The Extended I/O System checked the object directories of the local, global, and root jobs and was unable to find the specified logical name.
E$MEM	The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: CREATE$SEGMENT (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) S$LOOKUPCONNECTION
ENOTCONNECTION	The Extended I/O System found the logical name, but the name does not refer to a connection.
E$PARAM	This code indicates that the specified logical name is syntactically incorrect. Any one of the following problems can cause this error: • The logical name contains zero characters. This can only happen if there are no characters between the two colons (:). • The logical name contains more than 12 characters. • The logical name contains contains invalid characters.

The S$OPEN system call opens a connection so that your tasks can access the file through the connection.

```
CALL RQ$S$OPEN(connection, mode, num$buf, except$ptr);
```

INPUT PARAMETERS

connection — A WORD containing a token for the file connection to be opened.

mode — A BYTE telling how your task is going to use the connection. You should set the BYTE as follows:

Value	How Connection is Used
1	For reading only.
2	For writing only.
3	For both reading and writing.

num$buf — A BYTE containing the number of buffers that you want the Extended I/O System to allocate for this connection. This number must be between 0 and 255 (decimal).

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD in which the Extended I/O System will place the condition code.

DESCRIPTION

This system call performs four functions:

1. It creates the number of buffers requested.
2. It sets the connection's file pointer to zero. This is the pointer that tells the Extended I/O System where in the file to perform an operation.
3. It makes the Basic I/O System become ready to accept instructions from the Extended I/O System.

S$OPEN (continued)

DESCRIPTION (continued)

4. It starts reading ahead if the the number of buffers is greater than zero and the mode parameter includes reading.

Access Rights and Selecting a Mode

When you specify the mode, you must be accurate or err on the side of generosity. The Extended I/O System will not allow your tasks to read using a connection open for writing only. Nor will the Extended I/O System allow your tasks to write using a connection open for reading only. If you are not certain how the connection will be used, specify both reading and writing.

In the case of named files, the mode that you specify must match the access rights of the connection. (These are the access rights that the Extended I/O System assigned the connection when the connection was created.) For example, if your task attempts to open for reading a connection that has access for writing only, the Extended I/O System will return an exceptional condition code.

Selecting the Number of Buffers

The process of deciding how many buffers to allocate is based on three considerations -- the kind of file, memory, and performance. If your task is opening a connection to a stream file, the Extended I/O System always allocates zero buffers, regardless of the number of buffers you specify. However, if the connection is to a physical or named file, the Extended I/O System will allocate the number of buffers that you specify in the num$buf parameter.

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

The performance consideration is more complex. Up to a certain point, the more buffers you allocate, the faster your task can run. The actual break-even point, the point where more buffers don't improve performance, depends on many variables. Be aware that in order to overlap I/O with computation, you must specify at least two buffers.

If performance is important, and you have no idea how many buffers to specify, start with two. Once your task is running successfully, you can experiment empirically, adding or removing buffers until you have found the smallest number of buffers that allow your application to run as fast as possible.

DESCRIPTION (continued)

Alternatively, if performance is not at all important and memory is, use zero buffers.

Obtaining the Connection

The connection is subject to three constraints:

- It must already exist. The S$OPEN system call does not create a connection.
- It must be a file connection. You cannot open a device connection.
- The connection must have been created in the calling task's job. If the connection was created in a different job, use the S$ATTACH$FILE system call to obtain a new connection to the same file, and then open the new connection. If you attempt to use a file connection in a job other than that in which it was created, the Extended I/O System will return an E$SUPPORT condition code.

Compatibility with Basic I/O System

Refer to Appendix E to find out when you can use this system call in conjunction with the system calls of the Basic I/O System.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	Either the connection is a device connection, or the connection is already open.
E$FACCESS	The access rights embedded in the connection prohibit you from opening the file in the mode you have specified. This exceptional condition can arise only when the connection refers to a named data file or directory.
E$LIMIT	This code can be indicative of any of the following circumstances:

S$OPEN (continued)

CONDITION CODES

E$LIMIT (continued)

- The calling task's job is not an I/O job.
- The calling task's job, or the job's default user, is currently involved in more than 255 (decimal) I/O operations.

E$MEM — The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.

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

AGETFILE$STATUS (Basic I/O)
A$OPEN (Basic I/O)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)
ENABLE$DELETION (Nucleus)
GET$TYPE (Nucleus)
LOOKUP$OBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
SEND$CONTROL (Nucleus)
SEND$MESSAGE (Nucleus)
S$OPEN (Extended I/O)

ENOTCONNECTION — The connection parameter does not refer to a connection.

E$PARAM — The mode parameter is set to a value other than 1, 2, or 3.

E$SHARE — This code can be indicative of any of the following circumstances:

- Your task attempted to open a directory for writing only.
- Some task in your system is using the Basic I/O System to manipulate the file through another connection. That task requested that the Basic I/O System restrict the sharing of the file to certain modes. Your task is using a mode that precludes sharing the file.

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

The S$READ$MOVE reads a number of bytes from a file to a buffer. Your calling task must specify the connection, the number of bytes, and the buffer to receive the information.

```
bytes$read = RQ$S$READ$MOVE(connection, buf$ptr, bytes$desired,
                            except$ptr);
```

INPUT PARAMETERS

bytes$desired	A WORD containing the maximum number of bytes you want to read from the file.
connection	A WORD containing a token for the connection to the file. This connection must be open for reading or for both reading and writing, and the file pointer of the connection must point to the first byte to be read.

OUTPUT PARAMETERS

buf$ptr	A POINTER to a buffer that will receive the information that the Extended I/O System reads from the file.
bytes$read	A WORD containing the actual number of bytes that the Extended I/O System has read from the file. This number will always be equal to or less than the number of bytes desired. If it is less than the number of desired bytes, an end of file was encountered during the reading process.
except$ptr	A POINTER to a WORD in which the Extended I/O System will place a condition code.

DESCRIPTION

This system call reads a collection of contiguous bytes from the file associated with the connection. These bytes are placed in a buffer specified by the calling task.

S$READ$MOVE (continued)

DESCRIPTION (continued)

Creating the Buffer

The buf$ptr parameter tells the Extended I/O System where to place the bytes after they are read. Be aware of the following two stipulations relating to the buffer:

- You must create this buffer because the Extended I/O System does not. To create the buffer, use the CREATE$SEGMENT system call to obtain an iRMX 86 segment. This system call is discussed in the iRMX 86 NUCLEUS REFERENCE MANUAL. Alternatively, you can create a buffer during the compilation of your program.

- You must ensure that the buffer is long enough. If your task attempts to read more bytes than the buffer is capable of holding, the information immediately following the buffer could be overwritten. The Extended I/O System cannot sense when overwriting occurs, and consequently cannot advise your task when it does occur.

Number of Bytes Read

The number of bytes that your task requests is the maximum number of bytes that the Extended I/O System will place in the buffer. However, there are two circumstances under which the System will read fewer bytes.

- First, if the Extended I/O System detects an end of file before reading the number of bytes requested, it will return only those bytes preceding the end of file. The bytes$read parameter can be less than the bytes$desired parameter, and no exceptional condition will be indicated.

- Second, if an exceptional condition does occur during the reading operation, information in the buffer and the value of the bytes$read parameter are meaningless.

Access Control

If the connection is not opened for reading or both reading and writing, the Extended I/O System will return an exceptional condition. Also, if the connection was not created within the calling task's job, the Extended I/O System returns an E$SUPPORT exceptional condition.

DESCRIPTION (continued)

Specifying which Bytes Are Read

If your task is reading from a random-access file, your task must tell the Extended I/O System which bytes to read from the file. To do this, your task must position the connection's file pointer to the first of the bytes that you want to read. Use the S$SEEK system call to position the file pointer before your task invokes the S$READ$MOVE system call.

In contrast, if your task is reading from a sequential file, the Extended I/O System will maintain the connection's file pointer automatically.

Effects of Priority

The priority of the task invoking this system call can greatly affect the performance of the application system. For better performance, the priority of the invoking task should be lower than (numerically greater than) the priority of the Basic I/O System task that services the device containing the file. (To find out how to set priorities for application tasks, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL. To find out how to set priorities for Basic I/O System tasks, refer to the iRMX 86 CONFIGURATION GUIDE.) If the priority of the calling task is not lower than that of the Basic I/O System task, the I/O operation performed by this system call cannot be overlapped with computation or with other I/O operations.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception code can be caused by any of the following events: • The connection is not open for reading or for both reading and writing. • The connection is closed. • The connection was opened by the A$OPEN system call rather than the S$OPEN system call.
E$IO	An I/O error occurred during the reading operation.

S$READ$MOVE (continued)

CONDITION CODES (continued)

E$LIMIT	This code can be indicative of any of the following circumstances: • Either the calling task's job, or the job's default user, are currently involved in more than 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	The memory pool of the calling task's job does not currently contain 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: A$READ (Basic I/O) A$WRITE (Basic I/O) CREATE$SEGMENT (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus) S$READ$MOVE (Extended I/O)
ENOTCONNECTION	The connection parameter does not refer to a connection.
E$SPACE	One or more tasks of your application system are using the specified connection for both reading and writing. When this happens, the same buffers are used to accommodate both reading and writing. Consequently, before a task can use the buffer for reading, the Extended I/O System must ensure that the buffer is empty. This error was caused by full volume that prevented the Extended I/O System from emptying a writing buffer to the disk before your task's reading operation began.
E$SUPPORT	The connection parameter refers to a connection that was created by a task outside of the calling task's job.

The S$RENAME$FILE system call changes the name of a named file. It cannot be used for stream or physical files.

```
CALL RQ$S$RENAME$FILE(path$ptr, new$path$ptr, except$ptr);
```

INPUT PARAMETERS

path$ptr	A POINTER to a STRING that specifies the path for the file to be renamed. The syntax of this path is described in Chapter 5 of this manual. This path must refer to an existing file.
new$path$ptr	A POINTER to a STRING that specifies the new path for the file. This path must comply with the syntax and semantics of paths for named files as discussed in Chapter 5. Furthermore, this path cannot refer to an existing file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Extended I/O System will place a condition code.

DESCRIPTION

This system call, which can be used only with named files, allows your tasks to change the path for a file. You can rename directory files as well as data files.

Directory Files

Be aware that when you rename a directory, you are changing the paths for all files contained in the directory.

Restriction

If your task is renaming a file, the task can change any aspect of the file's path so long as the file remains on the same volume.

S$RENAME$FILE (continued)

DESCRIPTION (continued)

Access Control

In order to be able to rename a file, the default user of the calling task's job must have two kinds of access:

- Deletion access for the original file.
- Add-entry access for the file's new parent directory.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	This exception can be indicative of any of the following situations: • The device containing the specified file is in the process of being detached. • The calling task's job is not an I/O job. • The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device. • The two paths refer to different devices. • The calling task is attempting to rename a root directory.
E$DEVFD	When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$FACESS	This code can be returned under any of the following circumstances: • The default user of the calling task's job does not have add-entry access to the parent directory in the new$path$ptr parameter.

CONDITION CODES

E$FACCESS (continued)

- The default user of the calling task's job does not have delete access for the file to be renamed.

E$FEXIST — The new$path$ptr parameter refers to a file that already exists.

E$FNEXIST — This code is indicative of one of the following circumstances:

- Either some file in one of the specified paths, or the file being renamed, is marked for deletion.
- Some file in the specified path, or the file being, renamed does not exist.

E$FTYPE — The specified path is attempting to use a data file as a directory.

E$ILLVOL — When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.

E$IO — An I/O error occurred.

E$IOMEM — The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

E$LIMIT — This can be caused by any of the following conditions:

- While attempting to complete this system ¢all, the Extended I/O System created enough objects to exceed the object limit of the Basic I/O System job. Refer to the chapter of the iRMX 86 CONFIGURATION GUIDE that discusses the Basic I/O System.

S$RENAME$FILE (continued)

CONDITION CODES
E$LIMIT (continued)

- During the process of configuring your application system, someone assigned the Basic I/O System job a maximum priority that is too low. Specifically, the BIOS maximum priority is lower than either the DUIB priority or the DEVINFO priority. Refer to the iRMX 86 CONFIGURATION GUIDE for information regarding the BIOS maximum priority. Refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEMS for information regarding the DUIB and DEVINFO.

- Either the user object or the calling task's job is currently involved with more than 255 (decimal) I/O operations.

- The calling task's job is not an I/O job.

ELOGNAME$NEXIST At least one of the specified paths contains an explicit logical name, but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.

E$MEDIA The device containing the specified file is not on line.

E$MEM 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.

ENOPREFIX At least one of the specified paths contains no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.

ENOTCONFIGURED At least one of the following system calls was left out of the system during the configuration process:

A$ATTACH$FILE (Basic I/O)
A$PHYSICAL$ATTACH$DEVICE (Basic I/O)
A$RENAME$FILE (Basic I/O)
A$SPECIAL (Basic I/O)
CREATE$COMPOSITE (Nucleus)
CREATE$MAILBOX (Nucleus)

CONDITION CODES

ENOTCONFIGURED (continued)

CREATE$SEGMENT (Nucleus)
DELETE$COMPOSITE (Nucleus)
DISABLE$DELETION (Nucleus)
ENABLE$DELETION (Nucleus)
GET$DEFAULT$PREFIX (Basic I/O)
GET$TYPE (Nucleus)
LOOKUP$OBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
S$ATTACH$FILE (Extended I/O)
SEND$CONTROL (Nucleus)
SEND$MESSAGE (Nucleus)
SET$INTERRUPT (Nucleus)
S$RENAME$FILE (Extended I/O)
WAIT$INTERRUPT (Nucleus)

ENOTPREFIX — At least one of the specified paths contains a logical name that refers to an object that is neither a device connection nor a file connection.

ENOUSER — The calling task's job does not have a default user, or its default user is not a user object.

E$PARAM — This code can indicate either of the following conditions:

- At least one of the specified paths contains a logical name that is either longer than 12 characters or contains invalid characters.
- When your task invoked this system call, it forced the Extended I/O System to attempt the physical attachment of a device that had formerly been only logically attached. In the process of attempting to physically attach the device, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.

E$PREFIX$SYNTAX — At least one of the specified paths starts with a colon (:), indicating that it contains a logical name. But the Extended I/O System was unable to find a second colon to terminate the logical name.

E$SPACE — The volume is too full to allow the Extended I/O System to complete this operation.

S$RENAME$FILE (continued)

CONDITION CODES (continued)

E$SUPPORT	The calling task is attempting to rename a physical or stream file.

S$SEEK

Using the S$SEEK system call, your tasks can move the file pointer for any open physical- or named-file connection. This system call cannot be used with stream files.

```
CALL RQ$S$SEEK(connection, mode, hi$move$count, lo$move$count,
               except$ptr);
```

INPUT PARAMETERS

connection — A word containing a token for an open connection whose file pointer you wish to move.

hi$move$count
lo$move$count — These two WORDS combine to form a 32-bit integer that tells the Extended I/O System how, in bytes, to move the pointer.

mode — A BYTE containing a value that controls the nature of the movement of the file pointer. Any of the following values are valid:

Mode	Meaning
1	Move the pointer backward by the specified amount. If the move count is large enough to position the pointer past the beginning of the file, the pointer will be set to the first byte (position zero).
2	Set the pointer to the position specified by the move count. Position zero is the first position in the file. Moving the pointer beyond the end of the file is valid.
3	Move the file pointer forward by the specified amount. Moving the pointer beyond the end of file is valid.
4	First move the pointer to the end of the file and then move it backward by the specified amount. If the specified move count would position the pointer beyond the front of the file, the pointer will be set to the first byte in the file (position zero).

S$SEEK (continued)

OUTPUT PARAMETER

except$ptr	A POINTER to the WORD in which the Extended I/O System will place the condition code.

DESCRIPTION

When performing random I/O, your tasks must use this system call to position the file pointer before using the S$READ$MOVE, S$TRUNCATE$FILE, or S$WRITE$MOVE system calls. The location of the file pointer tells the Extended I/O System where in the file to begin reading, truncating, or writing information.

In contrast, if your tasks are performing sequential I/O on a file, they do not need to use this system call.

Access Control

There are two requirements that relate to access control. First, the connection must be open for reading only, writing only, or both reading and writing. If this is not the case, your task can use the S$OPEN system call to open the file.

The second access requirement is that the connection must have been created by a task within the calling task's job. If this is not the case, use the existing connection as a prefix, and have the calling task obtain a new connection by invoking the S$ATTACH$FILE system call. This newly created connection will satisfy the second requirement.

Special Considerations

As mentioned above, it is legitimate to position the file pointer beyond the end of file. If your task does this and then invokes the S$READ$MOVE system call, the Extended I/O System will behave as though the reading operation began at the end of file.

Also, it is possible to invoke the S$WRITE$MOVE system call with the file pointer beyond the end of the file. If your task does this, the Extended I/O System will attempt to expand the file. Be aware that, if the Extended I/O System does expand your file in this manner, the expanded portion of the file will contain random information.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	Either the connection parameter is not open, or it was opened by an A$OPEN rather than an S$OPEN.
E$IFDR	Your task is attempting to seek on a stream file. The S$SEEK system call can be used only with named and physical files.
E$IO	An I/O error occurred on the device containing the connection's file.
E$LIMIT	This code can be returned for any of the following reasons: • Either the calling task's job, or the job's default user, is currently involved with more than 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: A$SEEK (Basic I/O) A$WRITE (Basic I/O) CREATE$SEGMENT (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus) S$SEEK (Extended I/O)
ENOTCONNECTION	The connection parameter does not refer to a connection.
E$PARAM	This code can be returned for any of the following reasons:

S$SEEK (continued)

CONDITION CODES

E$PARAM (continued)

- The value of the mode parameter is not 1, 2, 3, or 4.
- The calling task was attempting to seek past the end of a physical file.

E$SPACE — This seek operation forced the Extended I/O System to attempt to empty the connection's buffer(s) by writing their contents to the volume. However, the volume was full and the Extended I/O System was unable to successfully empty the buffer(s).

E$SUPPORT — The connection parameter refers to a connection that was created by a task outside of the calling task's job.

The S$SPECIAL system call allows your tasks to perform functions that are peculiar to a specific device.

```
CALL RQ$S$SPECIAL(connection, function, data$ptr, iors$ptr,
                  except$ptr);
```

INPUT PARAMETERS

connection — A WORD containing a token for a connection to the file for which the special function is to be performed.

data$ptr — A POINTER to a parameter block that your task uses to supply the Extended I/O System with information. The contents and form of the parameter block depend upon the function being requested, so the form of the parameter block is described below, under the "DESCRIPTION" heading. If the function requires no parameter block, set the data$ptr to zero.

function — A WORD that specifies the special function being requested. Each function is described in detail under the "DESCRIPTION" heading, but the following table summarizes the values to be assigned to this parameter.

Value	Kind of Connection	Function
0	physical file on disk	format track
0	stream file	query
1	stream file	satisfy

Note that each function can be used with only certain kinds of files.

S$SPECIAL (continued)

OUTPUT PARAMETERS

except$ptr — A POINTER to a WORD in which the Extended I/O System will place the condition code.

iors$ptr — A POINTER to a structure of the form described below. The Extended I/O System uses this structure to return information that might be of use to the calling task. If you set this POINTER to zero, the Extended I/O System will not return the information. Be aware that this is relatively obscure information that most applications will not need.

```
iors$data STRUCTURE(
  actual        WORD,
  actual$fill   WORD,
  device        WORD,
  unit          BYTE,
  funct         BYTE,
  subfunct      WORD,
  lo$dev$loc    WORD,
  hi$dev$loc    WORD,
  buf$ptr       POINTER,
  count         WORD,
  count$fill    WORD,
  aux$ptr       POINTER)
```

where:

actual — is the number of bytes that were actually transferred during the special function.

actual$fill — is reserved for use in future versions of the Extended I/O System.

device — is the device number identifying the device. For an explanation of device numbers, refer to the iRMX 86 CONFIGURATION GUIDE.

unit — is the number of the unit that contains the file on which the special function is being performed. For information on unit numbers, refer to the iRMX 86 CONFIGURATION GUIDE.

OUTPUT PARAMETERS
iors$ptr (continued)

funct	is the function code indicating what operation was performed. In order to interpret this code, you must be intimately familiar with the device driver for the device containing the file on which the special function is being performed.
subfunct	is, in effect, an extension of the funct code.
lodevloc and hidevloc	The location on the device where the operation was performed.
buf$ptr	is the POINTER to the buffer used for this operation.
count	is the number of bytes that were transferred.
count$fill	is reserved for future use.
aux$ptr	is a POINTER to a data structure that contains information that you are not likely to need. If you really want to pursue this information, refer to the description of I/O Result Segments in the iRMX 86 CONFIGURATION GUIDE.

DESCRIPTION

This system call allows your tasks to perform functions that are closely dependent upon the kinds of devices your system uses. For this very reason, use of this system call will greatly reduce the degree of device independence supplied by the Extended I/O System. In other words, use this system call only if you must!

This system call allows your task to perform any of three special functions. The Extended I/O System decides which function to perform by examining the function parameter and the kind of connection provided in the connection parameter. The following table shows which function is performed for each combination of function code and kind of connection.

S$SPECIAL (continued)

DESCRIPTION (continued)

Function Parameter	Kind of Connection	Function Performed by the Extended I/O System
0	physical file on disk	Formats a track.
0	stream file	Provides information about stream file operations.
1	stream file	Artificially satisfies a stream file transaction.

The following three sections of this chapter explain each of these functions in detail.

Formatting a Track

In order to use the S$SPECIAL system call to format a track on a flexible diskette, the calling task must supply the following information:

connection — This parameter must contain a token for a connection to a physical file. This connection must be open for reading, writing or both.

function — Must be set to zero.

data$ptr — Must point to a STRUCTURE of the following form:

```
track$formatter   STRUCTURE(
   track$number      WORD,
   interleave        WORD,
   track$offset      WORD)
```

where:

track$number — contains the number of the track to be formatted. This value must lie between 0 and 76 decimal.

interleave — sets the interleaving factor for the track. The interleaving factor controls the number of physical sectors between consecutive logical sectors. If the interleave factor is 1, no physical sectors will be skipped.

DESCRIPTION (continued)

If it is 2, one sector will be skipped, and so on. Whatever value is provided in this parameter will be divided by the number of sectors per track, and the remainder will govern the number of physcial sectors to skip. An interleave factor of zero, or a factor that yields a remainder of zero, will produce unpredictable results.

track$offset contains the number of physical sectors to skip between the index mark and the first logical sector.

Formatting a track on a hard disk involves the same technique as is used for flexible diskettes. The only difference is that the track$formatter STRUCTURE looks like this:

```
track$formatter     STRUCTURE(
    track$number        WORD,
    interleave          WORD,
    track$offset        WORD,
    fill$char           WORD)
```

where:

track$number is the number of the track to be formatted. The Extended I/O System will accept values between 0 and 799 decimal.

interleave sets the interleaving factor for the track. The interleaving factor controls the number of physical sectors between consecutive logical sectors. If the interleave factor is 1, no physical sectors will be skipped. If it is 2, one sector will be skipped, and so on. Whatever value is provided in this parameter will be divided by the number of sectors per track, and the remainder will govern the number of physcial sectors to skip. An interleaving factor of zero, or a factor that yields a remainder of zero, will produce undefined results.

S$SPECIAL (continued)

DESCRIPTION (continued)

track$offset — the number of physical sectors to skip between the index mark and the first logical sector.

fill$char — the value that the device driver is to write into every byte of each sector. This applies only to iSBC 206 controllers.

Obtaining Information About Stream File Operations

Occasionally, a task using a stream file must find out what is being requested by another task using the same stream file. For example, the task reading a stream file might need to know how many bytes are being sent by a task writing to the same file. Tasks can obtain this kind of information by calling S$SPECIAL with the following information:

connection — A connection to the stream file.

function — Zero.

data$ptr — Any value whatsoever. This parameter is not used in performing this special function.

If a read request or a write request is queued at the file, the Extended I/O System will return information using the structure to which iors$ptr points. The following three fields contain valid information:

actual — the number of bytes already transferred toward satisfying the queued request.

count — number of bytes remainig to be transferred in satisfying the queued request.

buf$ptr — a POINTER to the memory location to be used for the next byte to be transferred.

funct — contains a value that indicates the purpose of the queued request. The value is 0 for read requests and 1 for write requests.

If no request is queued at the file, the Extended I/O System will queue the S$SPECIAL request for information at the file. This request will remain queued until a read or write request is issued. If, before a read or write request is issued, another S$SPECIAL request arrives, the Extended I/O System will cancel both S$SPECIAL requests and will return an E$CONTEXT exceptional condition code to the tasks that issued the call.

DESCRIPTION (continued)

Satisfying Stream File Transactions

As explained in Chapter 7 of this manual, stream files provide two tasks with the ability to communicate. When one task tries to read or write to a stream file, the task will not run again until the complementary task issues a matching request.

For example, suppose that Task A wants to read 512 bytes, but Task B wants to write only 256 bytes. Task A will stop running until Task B issues one or more requests which supply at least 256 bytes.

The S$SPECIAL system call provides tasks with the ability to force the Extended I/O System to consider a stream file transaction to be complete, even if the number of bytes written do not match the number of bytes read. To force this completion, a task must invoke the S$SPECIAL system call with the parameters set as follows:

connection	Must be a connection the stream file. This connection must be open for the operation that has not satisfied the matching requirement. For example, if the reading task wants to force the Extended I/O System to consider the transaction completed, the connection must be open for reading.
function	One.
data$ptr	Any value whatsoever. The Extended I/O System ignores this parameter for this special function.

After using the satisfy function of the S$SPECIAL system call, the only information that your task can obtain is the condition code returned by the Extended I/O System. If the task invoking S$SPECIAL system call has, in fact, already satisfied the transaction, the Extended I/O System will return an E$CONTEXT condition code.

Compatibility with Basic I/O System

Your application can use this system call to perform special functions on connections obtained from the Basic I/O System. Refer to Appendix E for a more general discussion of compatibility between the Extended and Basic I/O Systems.

S$SPECIAL (continued)

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The Extended I/O System can return this code for any of the following reasons: • Either the connection is not open, or it was opened by an A$OPEN rather than an S$OPEN. • The calling task is attempting to satisfy a stream file request, but there is no request queued at the stream file. • The calling task is querying a stream file, but the only request queued at the file is another query. The Extended I/O System removes both queries from the queue and returns this exception code.
E$IDDR	The function requested by the calling task is not supported for the device containing the specified file.
E$IFDR	The Extended I/O System does not support the requested function for the kind of file associated with the connection.
E$IO	An I/O error occurred while the Extended I/O System was attempting to perform the requested function.
E$LIMIT	The Extended I/O System can return this code for any of the following reasons: • Either the calling task's job or the job's default user is currently involved with more than 255 (decimal) I/O jobs. • The calling task's job is not an I/O job.
E$MEM	The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: A$SPECIAL (Basic I/O) CREATE$SEGMENT (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$TYPE (Nucleus)

CONDITION CODES

ENOTCONFIGURED (continued)

LOOKUP$OBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
SEND$CONTROL (Nucleus)
SEND$MESSAGE (Nucleus)
S$SPECIAL (Extended I/O)

ENOTCONNECTION	The connection parameter does not refer to a connection.
E$SPACE	During a formating operation, an attempt was made to seek past the end of a volume.
E$SUPPORT	The connection parameter refers to a connection that was created by a task outside of the calling task's job.

S$TRUNCATE$FILE

The S$TRUNCATE$FILE system call removes information from the end of a named data file. This system call can be used only with named files.

```
CALL RQ$S$TRUNCATE$FILE(connection, except$ptr);
```

INPUT PARAMETER

connection	A WORD containing a token for a connection to the named data file that is to be truncated. The file pointer of this connection tells the Extended I/O System where to truncate the file. The BYTE indicated by the pointer is the first byte to be dropped from the file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Extended I/O System will place a condition code.

DESCRIPTION

This system call applies to named data files only. When called, it truncates a file at the current setting of the file pointer and frees all space beyond the pointer.

End-of-File Considerations

If the pointer is at or beyond the end of file, no truncation will be performed.

Positioning the Pointer

Unless the file pointer is already where you want it, your task should use the S$SEEK system call to position the pointer before using the S$TRUNCATE$FILE system call.

DESCRIPTION (continued)

Interaction with Other Connections

The truncation will occur immediately, regardless of the status of other connections to the same file.

Access Requirements

There are three access requirements that relate to this system call. First the connection must be open for writing only or for both reading and writing. If this is not the case, your task can use the S$OPEN system call to open the connection.

Second, the connection must have update access for the file. Recall that the Extended I/O System computes a connection's access when the connection is created.

The third access requirement is that the connection must have been created by a task within the calling task's job. If this is not the case, use the existing connection as a prefix, and have the calling task invoke the S$ATTACH$FILE system call.

Compatibility with the Basic I/O System

Your tasks can use this system call in conjunction with files connections created by the Basic I/O System. However, you can only use this system call if the connection was opened by means of the S$OPEN (as opposed to A$OPEN) system call. Refer to Appendix E of this manual for a more general discussion of compatibility between the Extended and Basic I/O Systems.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	The Extended I/O System can return this code for any of the following reasons: • The connection is open in the wrong mode. It must be open for writing or for both reading and writing. • The connection is not open. • The connection was opened by an A$OPEN rather than an S$OPEN.

S$TRUNCATE$FILE (continued)

CONDITION CODES (continued)

E$IFDR	Your task is attempting to truncate a stream or physical file. The S$TRUNCATE$FILE system call can be used only on named files.
E$IO	An I/O error occurred on the device containing the file.
E$LIMIT	The Extended I/O System can return this code for any of the following reasons: • The calling task's job is not an I/O job. • Either the calling task's job, or the job's default user, is currently involved in more than 255 (decimal) I/O operations.
E$MEM	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.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: A$SEEK (Basic I/O) A$TRUNCATE$FILE (Basic I/O) A$WRITE (Basic I/O) CREATE$SEGMENT (Nucleus) DISABLE$DELETION (Nucleus) ENABLE$DELETION (Nucleus) GET$TYPE (Nucleus) LOOKUP$OBJECT (Nucleus) RECEIVE$CONTROL (Nucleus) RECEIVE$MESSAGE (Nucleus) SEND$CONTROL (Nucleus) SEND$MESSAGE (Nucleus) S$TRUNCATE$FILE (Extended I/O)
ENOTCONNECTION	The connection parameter does not refer to a connection.
E$SPACE	Before performing the truncation, the Extended I/O System must empty any buffers associated with the file. It can only empty these buffers by writing their contents to the file. However, the volume was full, so the Extended I/O System was unable to empty the buffers. This, in turn, prevented the Extended I/O System from truncating the file.
E$SUPPORT	The connection paramater refers to a connection that was created by a task outside the calling task's job.

The S$UNCATALOG$CONNECTION deletes a logical name from the object directory of a job. The task that invokes this system call must specify the logical name and the job.

```
CALL RQ$S$UNCATALOG$CONNECTION(job, log$name$ptr, except$ptr);
```

INPUT PARAMETERS

job	A WORD containing a token for a job. The Extended I/O System will delete the logical name from this job's object directory. If this parameter is set to zero, the Extended I/O System will remove the logical name from the object directory of the calling task's job.
log$name$ptr	A POINTER to a STRING that contains the logical name under which the connection is cataloged. The STRING can contain upper or lower case characters, but the Extended I/O System will convert them to upper case before looking up the logical name. The STRING should not contain any colons.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD in which the Extended I/O System will place the condition code.

DESCRIPTION

Your tasks should invoke this system call to undo the work of the S$CATALOG$CONNECTION system call. If the object directory of the specified job does not contain the specified logical name, the Extended I/O System will return an ELOGNAME$NEXIST exception code.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The job parameter refers to an object that does not exist.

S$UNCATALOG$CONNECTION (continued)

CONDITION CODES (continued)

E$LIMIT	The calling task's job is not an I/O job.
ELOGNAME$NEXIST	The Extended I/O System was not able to find the specified logical name.
E$MEM	The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	At least one of the following system calls was not incorporated into the system during the configuration process: CREATE$SEGMENT (Nucleus) UNCATALOG$CONNECTION (Extended I/O) UNCATALOG$OBJECT (Nucleus)
E$PARAM	The Extended I/O System detected a problem in the specified logical name. Any of the following circumstances can cause this condition code to be returned: • The STRING contains zero characters. • The STRING contains more than 12 characters. • The STRING contains characters that are not allowed within a logical name.
E$TYPE	The job parameter refers to an object other than a job.

The S$WRITE$MOVE system call writes a collection of bytes from a buffer to a file. Your task must specify the location of the buffer, the connection to the file, and the number of bytes to be written.

```
bytes$written = RQ$S$WRITE$MOVE(connection, buf$ptr, count,
                                except$ptr);
```

INPUT PARAMETERS

buf$ptr	A POINTER to a collection of contiguous bytes that are to be written to the specified file.
connection	A WORD containing a token for the connection to the file in which the information is to be written.
count	A WORD containing the number of bytes to be written from the buffer to the file.

OUTPUT PARAMETERS

bytes$written	A WORD containing the number of bytes that were actually written to the file. This number will always be equal to or less than the number specified in the count parameter.
except$ptr	A POINTER to a WORD in which the Extended I/O System will place a condition code.

DESCRIPTION

This system call causes the Extended I/O System to write the specified number of bytes from the buffer to the file.

Access Control

In order to write information into a file, the connection parameter must satisfy the following two requirements:

S$WRITE$MOVE (continued)

DESCRIPTION (continued)

- The connection must have been created by a task within the calling task's job. If this is not the case, the Extended I/O System will return an E$SUPPORT exception code.

- The connection must be open for writing or for both reading and writing.

Furthermore, if the file is a named data file, the access rights assigned to the connection must permit the kind of writing being performed. As far as the iRMX 86 System is concerned, there are two kinds of writing:

- Appending

  Appending is writing information only to the end of the file. The file becomes longer as more information is written. Whenever your task attempt to write information beyond the end of file via a connection that does not have append access, the Extended I/O System will return an exception code and will not write any data to the file.

- Updating

  Updating is writing over information already in the file. The length of the file is not changed because no new information can be added to the end of the file. Whenever your task attempts to write over information in a file via a connection that does not have update access, the Extended I/O System will not write any data to the file but will return an exception code.

The connection can have access rights for updating, appending, or both. For information regarding the process of assigning access to a connection, see the descriptions for the S$ATTACH$FILE and S$CREATE$FILE system calls.

Number of Bytes Acually Written

Occasionally, the Extended I/O System will actually write fewer bytes than requested by the calling task. This happens only under two circumstances. The first circumstance is when the Extended I/O System encounters an I/O error. Your task will be informed of this circumstance because the Extended I/O System will return an E$IO exception code.

The second circumstance is when the volume to which your task is writing becomes full. The Extended I/O System will inform your task of this condition by returning an E$SPACE exception code.

DESCRIPTION (continued)

Where the Bytes Are Written

The Extended I/O System writes the bytes starting at the location specified by the connection's file pointer. (The pointer indicates where the first byte is to be written.) As the Extended I/O System writes the bytes, it also updates the pointer. After writing operation is completed, the file pointer points to the byte immediately following the last byte written.

If your task must reposition the file pointer before writing, it can do so by using the S$SEEK system call.

If your task is using a connection that has append access, the task can start a writing operation beyond (rather than at) the end of file. The Extended I/O System will extend the file and perform the writing operation. Be aware that if your file is extended, the bytes in the extended section of the file contain random information until your task explicitly writes information into them. For example, if the end of file is at location 200 and your task positions the file pointer at 250 and begins writing, locations 200 through 249 contain undetermined information.

Effects of Priority

The priority of the task invoking this system call can greatly affect the performance of the application system. For better performance, the priority of the invoking task should be lower than (numerically greater than) the priority of the Basic I/O System task that services the device containing the file. (To find out how to set priorities for application tasks, refer to the iRMX 86 NUCLEUS REFERENCE MANUAL. To find out how to set priorities for Basic I/O System tasks, refer to the iRMX 86 CONFIGURATION GUIDE.) If the priority of the calling task is not lower than that of the Basic I/O System task, the I/O operation performed by this system call cannot be overlapped with computation or with other I/O operations.

CONDITON CODES

E$OK	No exceptional conditions.
E$CONTEXT	This condition can be caused by any of the following situations: • The connection is not open for writing. • The connection is not open.

S$WRITE$MOVE (continued)

CONDITION CODES

E$CONTEXT (continued)

- The connection was opened with A$OPEN rather than with S$OPEN.

E$IO — An I/O error occurred during the operation, and the Extended I/O System was unable to recover.

E$LIMIT — The Extended I/O System can return this exception code for any of the following reasons:

- The calling task's job is not an I/O job.
- The calling task's job, or the job's default user object, is currently involved in more than 255 (decimal) I/O operations.

E$MEM — The memory pool of the calling task's job does not currently contain a block of memory large enough to allow this system call to run to completion.

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

A$SEEK (Basic I/O)
A$WRITE (Basic I/O)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)
ENABLE$DELETION (Nucleus)
GET$TYPE (Nucleus)
LOOKUPOBJECT (Nucleus)
RECEIVE$CONTROL (Nucleus)
RECEIVE$MESSAGE (Nucleus)
SEND$CONTROL (Nucleus)
SEND$MESSAGE (Nucleus)
S$WRITE$MOVE (Extended I/O)

ENOTCONNECTION — The connection parameter does not refer to an existing connection.

E$PARAM — The calling task is attempting to write beyond the end of a physical file.

E$SPACE — The volume is full, and the Extended I/O System is unable to complete the requested writing operation.

E$SUPPORT — The connection paramter refers to a connection that was created by a task outside of the calling task's job.

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 form 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.

APPENDIX B. OBJECT TYPES AND RESOURCE REQUIREMENTS

Each iRMX 86 object type is known within the iRMX 86 system by means of a number called a type code. To allow you to refer to these codes symbolically, each code has a mnemonic name. The following list correlates the type of object, the mnemonic, and the type code:

OBJECT TYPE	MNEMONIC NAME	TYPE CODE
Job	T$JOB	1h
Task	T$TASK	2h
Mailbox	T$MAILBOX	3h
Semaphore	T$SEMAPHORE	4h
Segment	T$SEGMENT	6h
User	T$USER	100h
Connection	T$CONNECTION	101h
Logical Device	TLOGDEV	301h

RAM REQUIREMENTS

The following information is provided to help you estimate the amount of RAM that you will need in order to use the Extended I/O System. The discriptions that follow state explicitly from which pool the RAM is taken. You should use this information when deciding how large to make the memory pools of the jobs in your application. Be aware that this information applies only to the current release of the iRMX 86 Operating System and may shrink or grow in future releases.

ATTACHING A LOGICAL DEVICE

Each time one of your tasks uses the LOGICAL$ATTACH$DEVICE system call, the Extended I/O System uses 112 bytes of RAM from your job's pool and 100 bytes of RAM from the pool of the Extended I/O System job created during the configuration process. This RAM is in addition to the RAM required by the Basic I/O System for a device connection.

Both quantities of RAM are eventually returned to the memory pools from which they originated, but they are returned at different times. The memory taken from the Extended I/O System pool is returned only when the device is detached. In contrast, the memory taken from your job's pool is returned as soon the LOGICAL$ATTACH$DEVICE system call is finished running.

CREATING AN I/O JOB

Whenever one of your tasks creates an I/O job, the Extended I/O System uses 200 bytes of RAM from the pool of the I/O job being created. This RAM is in addition to the RAM used by the Nucleus to create the job. All of this memory will be returned to the pool of the parent job after the I/O job has been deleted.

In addition to the memory requirement, the CREATEIOJOB also requires five entries in the object directory of the I/O job being created. Refer to Appendix D to see how these entries are used.

OPENING A CONNECTION

Whenever one of your tasks uses the S$OPEN system call to open a file connection, the Extended I/O System uses some RAM from the pool of the calling job to create objects. The precise amount of RAM required depends upon whether the connection is opened for buffered I/O or nonbuffered I/O. If the connection is not buffered, the Extended I/O System uses 150 bytes of RAM. On the other hand, if the connection is buffered, you must use the following expression to compute the amount of RAM used as a function of the buffer size (S) and the number of buffers (N):

$$\text{number of bytes} = 150 + N(S + 80)$$

Regardless of whether the connection is buffered, all RAM is returned to the memory pool when the connection is closed or deleted.

OTHER RAM REQUIREMENTS

For system calls other than those discussed above, the Extended I/O System has varying memory requirements. However, you can safely assume that the Extended I/O System will use no more than 300 bytes of your job's RAM during each system call. This RAM will be returned to your job's pool as soon as each system call finishes running.

OBJECT COUNTS

Because each job has a maximum number of objects that it can own, you should be aware of the number of objects that the Extended I/O System creates while executing system calls. You can assume that the Extended I/O System creates no more than 10 objects during the execution of any system call.

Furthermore, except in a few cases, all of these objects are deleted before the system call has finished running. The few exceptions are the system calls that explicitly create objects at the request of your application tasks. Two examples of system calls that explicitly create objects are the S$ATTACH$FILE system call (which creates a device connection) and the LOGICAL$ATTACH$DEVICE system call (which creates a device connection).

APPENDIX C. CONDITION CODES

The iRMX 86 Extended I/O System uses condition codes to inform your tasks of any problems that occur during the execution of a system call. If no problems occur and the system call runs to completion, the Extended I/O System returns an E$OK condition code. Otherwise, the Extended I/O System returns an exceptional condition code.

The meaning of a specific exceptional condition code depends upon the system call that returns the code. For this reason, this appendix does not list any interpretations.

The purpose of this appendix is to provide you with the numeric value associated with each condition code that the Extended I/O System can return. To use the exception 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 the condition code (as described in Chapter 8 of this manual) to 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 indicative of a programming error, and one for exception codes indicative of an environmental error.

NORMAL CONDITION CODE

NAME OF CONDITION	HEXADECIMAL VALUE
E$OK	0h

PROGRAMMING EXCEPTION CODES

NAME OF CONDITION	HEXADECIMAL VALUE
E$IFDR	8020h
E$NOPREFIX	8022h
ENOTCONNECTION	8042h
ENOTDEVICE	8041h
ENOTPREFIX	8040h
ENOTSUPPORTED	8005h
E$NOUSER	8021h
E$OVERFLOW	8001h
E$PARAM	8004h
E$TYPE	8002h
E$ZERO$DIVIDE	8000h

ENVIRONMENTAL EXCEPTION CODES

NAME OF CONDITION	HEXADECIMAL VALUE
E$BUSY	3h
E$CANNOT$CLOSE	41h
E$CONTEXT	5h
E$DEVFD	22h
EDIREND	25h
E$EMPTY$ENTRY	24h
E$EXIST	6h
E$FACCESS	26h
E$FEXIST	20h
E$FLUSHING	2Ch
E$FNEXIST	21h
E$FTYPE	27h
E$IDDR	2Ah
E$IO	2Bh
E$IOMEM	42h
E$LIMIT	4h
ELOGNAME$NEXIST	45h
E$MEDIA	44h
E$MEM	2h
ENOTCONFIGURED	8h
E$PREFIX$SYNTAX	40h
E$SHARE	28h
E$SPACE	29h
E$STATE	7h
E$SUPPORT	23h
E$TIME	1h

APPENDIX D. USE OF OBJECT DIRECTORIES BY THE EXTENDED I/O SYSTEM

The Extended I/O System catalogs entries in the object directory of each I/O job and in the object directory of the system's root job. This appendix provides a list of the names that the Extended I/O System uses. The reason for providing this list is to allow you to ensure that you do not accidentally redefine any of these names.

RQGLOBAL	The Extended I/O System uses this name to identify the global job for each I/O job. Whenever you create an I/O job, the Extended I/O System automatically catalogs the token for the global job in the object directory of the I/O job. If you wish to redefine this name, you may. But be aware that doing so may alter the interpretation of any logical names that are cataloged in the object directory of your job's global job.
R?IOJOB	Whenever you create an I/O job, the Extended I/O System catalogs an object under this name in the object directory of the I/O job. Do not redefine this name!
R?MESSAGE	Whenever you create an I/O job, the Extended I/O System catalogs an object under this name in the object directory of the I/O job. Do not redefine this name!
R?USER	The Extended I/O System uses this name to catalog the default user for each I/O job. Be aware that if you use the CATALOG$OBJECT system call to alter the definition associated with this name, you will change your job's default user. Furthermore, if you catalog an object other than a user object under this name, the Extended I/O System will generate exceptional conditions codes whenever your tasks attempt to access a named file.
$	The Extended I/O System uses this name to catalog the default prefix for each I/O job. If you modify the definition associated with this name by invoking the CATALOG$OBJECT system call, you will be changing the job's default prefix. Furthermore, if you catalog an object other than a device connection or a file connection under this name, the Extended I/O System will generate an exceptional condition code whenever you attempt to use the default prefix.

With the exception of RQGLOBAL and $, you should not use the CATALOG$OBJECT system call to modify any of the definitions described above. If you do change any of them, you may cause the Extended I/O System to behave in an unexpected, unpredictable, and undesirable manner.

The Extended I/O System uses object directories for two other purposes:

- Whenever you use the CATALOG$CONNECTION system call to define a logical name for a connection, the Extended I/O System catalogs the connection in the object directory of the job that you specify.

- Whenever you use the LOGICAL$ATTACH$DEVICE system call, the Extended I/O System catalogs the device connection in the object directory of the system's root job.

APPENDIX E. COMPATIBILITIES BETWEEN THE TWO SYSTEMS

Many of the system calls in one I/O System have counterparts in the other I/O System. For example, the A$CREATE$FILE system call of the Basic I/O System performs a function analogous to the S$CREATE$FILE system call of the Extended I/O System. So it is reasonable to ask if connections created by one system can be used by the other.

The answer is yes, unless the connection is open. For example, your application system can use the S$CREATE$FILE system call of the Extended I/O System to create a file and obtain a connection to the file. Because the connection is not open, your application system can use the connection with any system call of the Basic I/O System that does not require an open connection. For instance, the connection can be used with A$RENAME$FILE or with AGETFILE$STATUS because neither of these system calls require that the connection be open. However, the connection cannot be used with A$READ or A$WRITE because both of these system calls require that the connection be open.

The same restriction applies if the connection is created using the Basic I/O System. The connection can be used with any system call of the Extended I/O System so long as the system call does not require an open connection.

In general, you can create, delete, check status, or attach using either kind of system call. But once you have opened the connection, you must use a read, write, truncate, or special-function system call provided by the I/O System that you used to open the connection. Then, once you have closed the connection, you can again use system calls from either I/O System.

INDEX

For most topics with multiple-page references, the primary reference is underscored.

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 _____

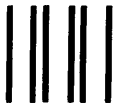
ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

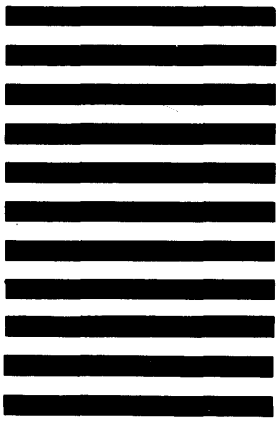
Please check here if you require a written reply. ☐

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

Printed in U.S.A.