iRMX 86™ BASIC I/O SYSTEM REFERENCE MANUAL

Order Number: 9803123-04

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

REV.	REVISION HISTORY	PRINT DATE
-01	Original Issue	4/80
-02	Application Loader added and unimplemented system calls removed.	11/80
-03	Application Loader information removed. Changes made to reflect Release 3 of the iRMX 86 Operating System.	5/81
-04	Exception codes updated. Changes reflect Release 4 of iRMX 86. Change bars mark technical changes.	10/81

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	Intelevision	Micromap
CREDIT	Intellec	Multibus
i	iRMX	Multimodule
ICE	iSBC	Plug-A-Bubble
iCS	iSBX	PROMPT
im	Library Manager	Promware
INSITE	MCS	RMX/80
Intel	Megachassis	System 2000
Intel	Micromainframe	UPI
		μScope

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

A492/1181/ 2K DD

PREFACE

This manual documents the Basic 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 as a quick reference to system calls, providing detailed descriptions of those system calls available to application programmers. Other system calls, which are reserved for system programmers, are discussed generally, but only to give an overview of Basic I/O System operation. The reserved system calls are discussed in detail in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

READER LEVEL

This manual is intended for application programmers who are familiar with the concepts and terminology introduced in the iRMX 86 NUCLEUS REFERENCE MANUAL and with the PL/M-86 programming language.

CONVENTIONS

Throughout this manual, system calls are named using a generic shorthand (such as CREATE$FILE instead of RQ$A$CREATE$FILE). The actual PL/M-86 external-procedure names used to invoke these operations are shown only in Chapter 8, which lists the detailed calling sequences.

Chapter 8 of this manual, which contains detailed descriptions of the system calls, lists only the PL/M-86 calling sequences. The system calls can be invoked from assembly language, but in order to do so, you must obey the PL/M-86 calling conventions, which are discussed in the iRMX 86 PROGRAMMING TECHNIQUES manual.

RELATED PUBLICATIONS

The following manuals provide additional background and reference information.

Manual	Number
Introduction to the iRMX™ 86 Operating System	9803124
iRMX™ 86 Nucleus Reference Manual	9803122
iRMX™ 86 Debugger Reference Manual	143323
iRMX™ 86 Terminal Handler Reference Manual	143324
iRMX™ 86 System Programmer's Reference Manual	142721
iRMX™ 86 Programming Techniques Manual	142982
Guide to Writing Device Drivers for the iRMX™ 86 I/O System	142926
iRMX™ 86 Extended I/O System Reference Manual	143308
iRMX™ 86 Configuration Guide	9803126
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	9803478
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

CONTENTS (continued)

CONTENTS (continued)

FIGURES

CHAPTER 1. ORGANIZATION

This manual is divided into eight chapters. Some of the chapters contain introductory or overview material which you do not need to read if you are already familiar with the subsystems or if you have used this manual before. Other chapters contain reference material which you will refer to as you write your application tasks. You can use this chapter to determine which of the other chapters you need to read. 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 features of the Basic I/O System. You should read this chapter if you are going through the manual for the first time or if you have had very little previous exposure to the Basic I/O System.
Chapter 3	This chapter explains some basic terminology associated with the Basic 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 Basic I/O System.
Chapter 4	This chapter describes how to use the asynchronous system calls that are included in the Basic I/O System. You should read this chapter before you write tasks that make asynchronous system calls.
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 of the Basic I/O System, in alphabetical order. When writing your application tasks, you should refer to this chapter for specific information about the format and parameters of the system calls.

CHAPTER 2. FEATURES OF THE BASIC I/O SYSTEM

Because the iRMX 86 Operating System is designed 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:

- Asynchronous Operation
- Device Independence
- Support for Many Kinds of Devices
- Three Distinct Kinds of Files
- File Sharing and Access Control
- Separation of File Lookup and File Open Operations
- Control over Fragmentation of Files

The purpose of this chapter is to briefly explain each of these features and to familiarize you with the terminology of the Basic I/O System.

ASYNCHRONOUS OPERATION

When you examine the system call chapter of this manual, you will find that the system calls can be divided into two categories according to their names. The first category consists of system calls having the names of the form:

```
RQ$XXXXX
```

where XXXXX is a brief description of what the system call does. The second category consists of system calls having names of the form:

```
RQ$A$XXXXX
```

System calls of the first category, without the A, are synchronous calls. They begin running as soon as your application invokes them, and continue running until they detect an error or accomplish everything they must do. Then they return control to your application. In other words, synchronous calls act like subroutines.

System calls of the second category (those with the A) are called asynchronous because they accomplish their objectives by using tasks that run concurrently with your application. This allows your application to accomplish some work while the Basic I/O System deals with mechanical devices.

For more detail on the behavior of asynchronous system calls, refer to Chapter 4.

DEVICE INDEPENDENCE

The Basic 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 can 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 Basic I/O System allows you to create an application that can log the events on any device rather than just one. When the event application is running and circumstances force an operator to reroute logging from, for instance, the teletypewriter to the line printer, your application can easily comply.

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

SUPPORT FOR MANY KINDS OF DEVICES

Although your application can be device independent, the Basic I/O System must be able to communicate with a wide variety of devices. In order to connect a particular device to the Basic I/O System, you must have a device driver (a collection of software procedures) designed especially for the device being connected.

The Basic I/O System currently provides drivers for the following devices:

- iSBC 204 Single Density Flexible Disk Controller
- iSBC 206 Hard Disk Controller
- iSBC 254 Bubble Memory
- iSBC 215 Winchester Hard Disk Controller
- iSBC 218 Multimodule Flexible Disk Controller
- Byte Bucket
- 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. 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 Basic I/O System, refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

THREE DISTINCT KINDS OF FILES

Although all files in the Basic I/O System are byte (as opposed to record) oriented, the System provides you with three kind of files:

NAMED FILES

Named files are intended for use with random-access, secondary storage devices such as disks, diskettes, and bubble memories. They 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 a good starting place for building custom access methods such as ISAM (indexed sequential access method).

For more detailed information regarding named files, refer to Chapter 5.

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 it as though it were a string of bytes. Also, physical files do not provide access control.

This more basic relationship with a device provides your application with flexibility. For example, your application can interpret volumes created on other systems by using physical files.

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, see Chapter 6.

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, see Chapter 7.

FILE SHARING AND ACCESS CONTROL

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

FILE SHARING

In a multitasking system it is often useful to have several tasks manipulating a file simultaneously. Consider, for example, 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 Operating System allows multiple tasks to concurrently access the same file.

For more detailed information about sharing files, see Chapters 5, 6, and 7.

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 reserve to himself the ability to delete his files, while allowing people in his department to write and read his file, and people in other departments to only read the files. The Basic I/O named files provide your applications with this kind of access control.

For more detailed information regarding access control, refer to Chapter 5.

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 Basic I/O System avoids this by using a special type of object (called a connection) to represent the bond between the file and a program.

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

There are several other benefits associated with connection objects. In the case of named files, connections embody the access rights to the file. This means that access need only be computed once (when the connection is created) rather than each time the file is opened.

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

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

CONTROL OVER FRAGMENTATION OF FILES

The Basic I/O System allows your application to specify the granularity of each mass storage file. This lets you trade faster I/O for more efficient use of space on the mass storage device.

When information is stored on a mass storage device, space is allocated in chunks rather than one byte at a time. These chunks (called granules) can be large or small, but all granules within one file must be of the same size. This size is called the file granularity. The Basic I/O System allows your application to specify the granularity of each file that it creates.

For a detailed explanation of the benefits of control over file fragmentation, see the INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM.

CHAPTER 3. BASIC TERMINOLOGY

There are five concepts that you must understand if you wish to use the Basic I/O System. These concepts are:

- system programmers
- devices
- volumes
- files
- connections

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 involes 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 actions to be performed by system programs written by system programmers. Because of the broad effect of these functions, 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 Basic 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 Basic 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 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 Basic I/O System considers a file to be information.

The Basic 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 2 and in detailed terms in Chapters 5, 6, and 7.

Regardless of the kind of file, the Basic 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 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 Basic 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.

Applications typically obtain device connections by invoking a system program written by your system programmer. This program passes the connection for the desired device to the calling application program.

Once the device has been attached, the only way to refer to the device is by the connection. If your application catalogs the device connection in your job's object directory under the name of the device, all of the tasks in the job will be able to refer to the device. This is one of several ways of making the device connection available to tasks.

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 Basic 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 Basic I/O System with information describing the bond. This information includes:

- a file pointer

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

- an open-mode indicator

  The Basic I/O System sets this variable when your application calls the A$OPEN system call to open this connection to the file. The open-mode indicator tells the Basic I/O System how your application 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.

- a share-mode indicator

  The Basic I/O System sets this variable when your application calls the A$OPEN system call. The share-mode indicator controls how other connections can share the file with the connection being opened. This variable can take on any of four values: no sharing whatsoever, share with readers only, share with writers only, share with readers or writers.

CHAPTER 4. ASYNCHRONOUS SYSTEM CALLS

Each asynchronous system call has two parts -- one sequential, and one concurrent. As you read the descriptions of the two parts, refer to Figure 4-1 to see how the parts relate.

- the sequential part

  The sequential part behaves in much the same way as the fully synchronous system calls in Chapter 2. Its purpose is to verify parameters, check conditions, and prepare the concurrent part of the system call. The sequential part then returns control to your application.

- the concurrent part

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

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

Let's look at a brief example showing how your application can use asynchronous calls. Suppose your application requires some information that is stored on disk. The application issues the A$READ system call to have the Basic I/O System read the information into memory. Let's trace the action one step at a time:

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

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

3. If the sequential part of the call detects a problem, it signals an exception and returns control to your application. It does not make ready the Basic I/O System task to perform the reading function.

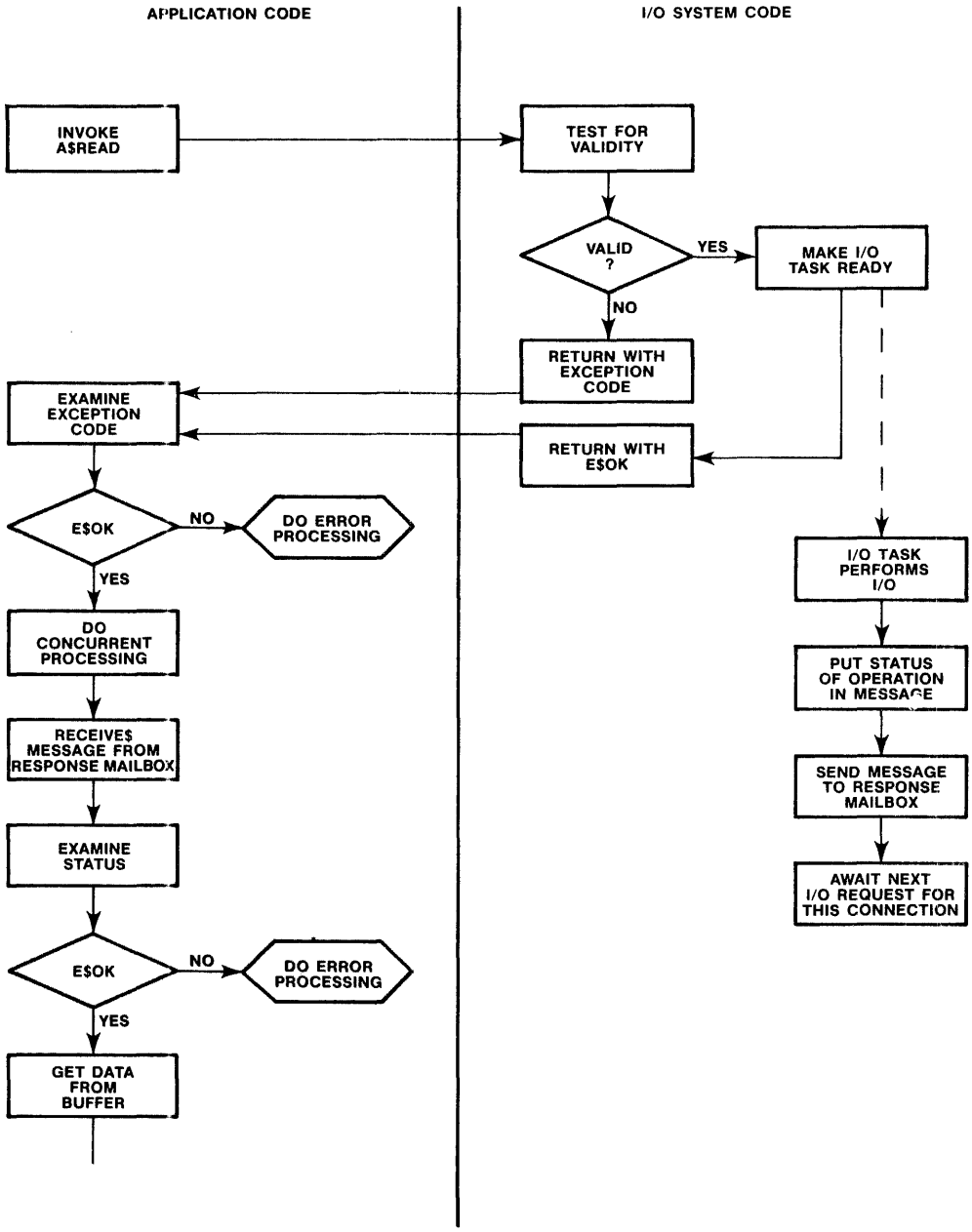

Figure 4-1. Concurrent Behavior of an Asynchronous System Call

4. Your application receives control. Its behavior at this point depends on the condition code returned by the sequential part of the system call. Therefore, the application tests the condition code. If the code is E$OK, the application continues running until it must have the information from the disk. It is at this point that your application can take advantage of the asynchronous and concurrent behavior of the Basic I/O System.

   For example, your application can implement double (or multiple) buffering by issuing another (or several) A$READ system call(s) while waiting for the first call to complete running. Alternatively, your application can use this overlapping processing to perform computations. The point is that you can decide what you want your application to do while the asynchronous system call is running.

   On the other hand, if your application finds that the condition code returned from the sequential part of the system call is other than E$OK, the application can assume that the Basic I/O System did not make ready a task to perform the function.

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

5. Your application now must have the information. Before taking the information from the buffer, you application must verify that the concurrent part of the A$READ system call ran successfully. The application issues a RECEIVE$MESSAGE system call to check the response mailbox that the application specified when it invoked the A$READ system call.

   By using the RECEIVE$MESSAGE system call, the application obtains a segment that contains, among other things, a completion code for the concurrent part of the A$READ system call. If this completion code is E$OK, then the reading operation was successful, and the application can get the data from the buffer. On the other hand, if the code is not E$OK, the application should analyze the code and attempt to determine why the reading operation was not successful.

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

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

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

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

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

- Whenever the concurrent part of a system call runs, the Basic I/O System signals its completion by sending an object to the response mailbox. The precise nature of the object depends upon which system call your application invoked. You can find out what kind of object comes back from a particular system call by looking up the call in Chapter 8 of this manual.

- Whenever the Basic I/O System returns a segment to your application's response mailbox, your application must delete the segment when it is no longer needed. The Basic I/O System draws memory for such segments from your job's memory pool, so if your application fails to delete the segment, your job may run short of memory.

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. Named file trees also must fit on a single volume.

Each file tree consists of two categories of files -- data files and directories. Data files (which are 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. Figure 5-1 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.

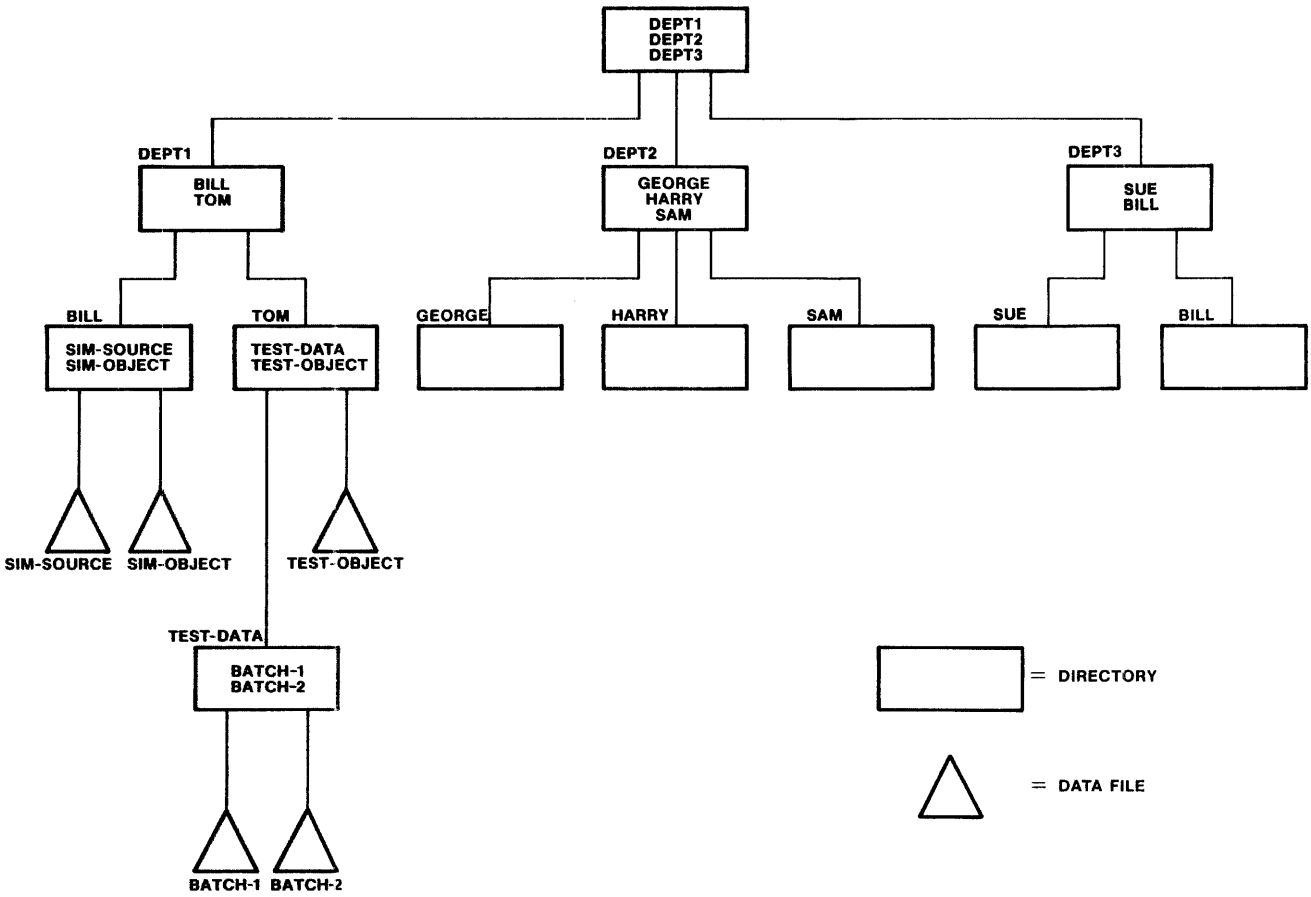

Figure 5-1. Example of a Named-File Tree

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." And 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. 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

Whenever your application manipulates either kind of named file, the application must tell the Basic I/O System which file is to be manipulated. There are several ways to specify a particular named file to the Basic I/O System, all of which involve connections and paths.

CONNECTIONS

Once you have a connection to a particular named file, you can use the connection as the PREFIX parameter of any system call. If, in the same call, you set the SUBPATH parameter to zero, the Basic I/O System will ignore the SUBPATH and use only the PREFIX to find the file.

PATHS

If you do not have a connection to the file you can specify the file by using its path. To do this, build an iRMX 86 string of the form described in the opening pages of Chapter 8. (An iRMX 86 string is a representation of a character string. To represent a string of n characters, you must use n+1 consecutive bytes. The first byte contains the character count, n. The following n bytes contain the ASCII codes for the characters in the same order as the string.) This string is called a path name. Then use a pointer to this path name as the SUBPATH parameter in the system call, and use the device connection as the PREFIX parameter in the system call.

For example, if your named file tree is on Drive 1, and it has the path name DEPT_2/HARRY/TEST_RESULTS, you can specify the file by using the device connection for Drive 1 as the PREFIX parameter and a pointer to the path name as the SUBPATH parameter.

PREFIX AND SUBPATH

Once your application has obtained a connection to a directory file within a named file tree, the application can use that connection as a basis for reaching all files that descend from the directory.

For example, referring again to Figure 5-1, suppose your application has a connection to Directory DEPT_1/TOM. The application can refer to Data File BATCH_1 by using both the PREFIX and the SUBPATH parameters. The application should use the connection to Directory DEPT_1/TOM as the PREFIX, and it should use a pointer to a subpath name as the SUBPATH. The subpath name is a string that connects Directory DEPT_1/TOM to Data File BATCH_1. For this example, the subpath name is TEST_DATA/BATCH_1.

DEFAULT PREFIX

Within one iRMX 86 job, most references to a named file tree are generally confined to one branch of the tree. For example, in Figure 5-1, Tom will usually access the files in his directory more frequently than files outside of his directory. Recognizing this clustering, the Basic I/O System provides the notion of default prefix.

The Basic I/O System allows your application to specify one default prefix for each iRMX 86 job. A default prefix is a connection to a directory at the head of the most commonly used branch in your named file tree. For instance, in Figure 5-1, Tom's application would probably use a connection to Directory DEPT_1/TOM as the default prefix. To use the default prefix, the application sets the PREFIX parameter to zero.

A default prefix provides a job with two advantages. First, by providing a reference point within a named file tree, it allows your application to use subpath names instead of path names. If your tree is several levels deep, this can save coding time during development. Second, and more significantly, a default prefix provides a means of writing generalized application code that can work at any of several locations within a tree.

Consider an example. Suppose that an assembler (implemented as an iRMX 86 job) uses a default prefix to find a location in a named file tree. The assembler could then use a subpath name of TEMP to find or create a temporary work file. Before an application invokes the assembler, it sets the default prefix of the assembler job to a directory in the application's named file tree. This allows more than one job to invoke the assembler concurrently without the risk of sharing temporary files.

The Basic I/O System keeps track of a job's default prefix by using the job's object directory. Whenever your tasks use the SET$DEFAULT$PREFIX system call to specify a connection as being the default, the Basic I/O System catalogs the connection under the name $ in the job's object directory.

USERS AND ACCESS RIGHTS

Named files provide 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 Basic I/O System implements an iRMX 86 object type called a user object, but 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 enter information, 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 its members. 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. To do this, your application must employ a special group, called World, that includes all users.

User Objects

The Basic 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 attempts to gain access to or create a named file, the application must present a user object to the Basic I/O System. The Basic I/O System then uses this 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 automatic 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). 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 Basic 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 to create the object and pass 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 provides three system calls for creating, deleting and inspecting user objects. These calls are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

Default Users

Generally, most of the I/O operations performed within a particular iRMX_86 job are performed on behalf of one user object. Recognizing this, the Basic I/O System allows your application to designate one default user per job. Whenever your application invokes a Basic I/O System call on behalf of the default user, the application can use zero as the token for the USER parameter. The Basic I/O System will recognize the zero as referring to the default user.

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 a 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 Basic I/O System calls having a zero user object will be performed on behalf of the default user.

The Basic I/O System provides two system calls to manipulate a job's default user. The GET$DEFAULT$USER and SET$DEFAULT$USER calls are both described in the system call chapter of this manual.

The Basic I/O System uses the job's object directory to keep track of the job's default user. Whenever one of your tasks sets or gets a default user, the Basic I/O either catalogs or looks up the default user entry in the object directory. The Basic I/O System uses the name R?USER to refer to the default user. To prevent problems, you should consider R?USER to be a reserved name, and you should avoid using it.

ACCESS RIGHTS

For each named file (directory or data file), the Basic 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 an encoded 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 Basic 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 A$DELETE$FILE and rename the file with A$RENAME$FILE.
Read	The ability to read the file with A$READ.
Append	The ability to add information to the end of the file with A$WRITE.
Update	The ability to change information in the file with A$WRITE or drop information with A$TRUNCATE.

The kinds of access rights available for directory files are:

Delete	The ability to delete the directory file with A$DELETE$FILE.
Display	The ability to obtain the contents of directory files with A$READ or A$GET$DIRECTORY$ENTRY.
Add Entry	The ability to add files to the directory with A$CREATE$FILE, A$CREATE$DIRECTORY, or A$RENAME$FILE.
Change Entry	The ability to change the access rights of the files in the directory with A$CHANGE$ACCESS.

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

When an application creates a named file, the application uses the A$CREATE$FILE system call. Two of the parameters of this call are USER and ACCESS. When the Basic I/O System actually builds the file, it initializes the access list with a single entry consisting of the id of the user who invoked A$CREATE$FILE and the access he specified in the call. 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 A$CHANGE$ACCESS system call to modify the file's access list without being granted explicit permission to do so.

Computing Access

Whenever an application attempts to access a named file, the application must supply the Basic I/O System with a user object. The Basic 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 Basic I/O System computes the access by "or"ing together the access of each matching entry.

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

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

Now suppose that the application presents a user object having an id list of

```
042A
8B01
FFFF
```

The Basic 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 Basic 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 Basic I/O System computes access only under two circumstances. The first circumstance is the creation of a connection. Whenever an application creates a connection (by using the A$CREATE$FILE, A$CREATE$DIRECTORY, or A$ATTACH$FILE system calls), the application presents a user object to the Basic I/O System. 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 attempts to manipulate the file via the connection, the Basic 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 embedded in the connection.

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

If an invocation of A$DELETE$FILE or A$CHANGE$ACCESS does contain the null subpath, the Basic I/O System will use the access associated with the PREFIX to decide whether or not to perform the function requested in the system call.

NOTE

If a system call invocation contains a subpath parameter other than the null subpath, the Basic 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.

Access at Time of Creation

Whenever your application creates a named file (either data or directory), the application presents two access-related parameters to the Basic I/O System. One of the parameters is a user object. The Basic I/O System uses this object to "brand" the file as being owned by a specific user.

The second access-related parameter is called ACCESS. This parameter governs the owner's access rights. The kinds of access from which the application can choose depend upon whether a data file or a directory is being created. These rights are discussed in the "Access Rights" section of this chapter.

Granting Access to Other Users

When an application 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 A$CHANGE$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 creates a file, it uses a group's user object rather than its own user object, the group is the owner of the file. Using A$CHANGE$ACCESS, any user in the group can change the kind of access granted to the group, or the kind of access granted to any other accessor of the file.

SYSTEM CALLS FOR NAMED FILES

There are 31 system calls that relate to iRMX 86 named files. Some of these calls are useful for both data and directory files, some only for one kind of file, and some (such as CREATE$USER) don't relate to either kind of file.

The following sections briefly explain the purpose of each of the 31 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

There are six system calls that relate to obtaining and deleting connections.

- A$CREATE$FILE

  This call applies only to data files. Your application must use this call to create a new data file, and it can use this call to obtain a connection to an existing data file. If the application uses this call to create a new file, the Basic I/O System automatically adds an entry in the parent directory for this new file.

- A$CREATE$DIRECTORY

  This call applies only to directory files. Your application must use this call to create a new directory file. The call cannot be used to obtain a connection to an existing directory. The Basic I/O System automatically adds an entry in the parent directory for this new directory.

- A$ATTACH$FILE

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

- A$DELETE$CONNECTION

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

- A$ATTACH$DEVICE

  This call does not directly apply to either data or directory files. Your application uses this call to obtain a connection to a device. 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.

- A$DETACH$DEVICE

  This call does not directly apply to either data or directory files. Your application uses this call to delete a connection to a device. This system call is explained in detail in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

USER OBJECTS

There are five calls directly related to user objects. None of these calls is specifically related to data or directory files. The calls are:

- CREATE$USER

  This call is used to create a user object. Since this call is generally invoked only by system program, 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 can use this call to establish a default user for any iRMX 86 job. This call is described in Chapter 8 of this manual.

- GET$DEFAULT$USER

  Your application can use this call to ascertain the default user for any iRMX 86 job. This call is described in Chapter 8 of this manual.

DEFAULT PREFIXES

There are two calls that relate to default prefixes, and both are described in detail in Chapter 8 of this manual. Neither of these calls relates directly to data files or directory files. The calls are:

- SET$DEFAULT$PREFIX

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

- GET$DEFAULT$PREFIX

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

MANIPULATING DATA

There are six system calls that allow you 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:

- A$OPEN

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

- A$CLOSE

  This call applies to both data and directory files. After your application 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 Basic I/O System close it when the connection is deleted, but there is an advantage to closing connections when they are not being used.

  This advantage derives from the fact that, when a connection is shared between two or more applications, some of the applications can place restrictions on the manner of sharing. For instance, an application can specify sharing with writers only. By closing connections, your application can improve the likelihood that the connections can be used by other applications.

- A$SEEK

  This system call applies to both data and directory files. Whenever your application reads, writes, or truncates a file, the application must tell the Basic I/O System the location in the file where the operation is to take place. To do this, your application uses the A$SEEK system call to position the file pointer of the file connection. The A$SEEK system call requires that the file connection be open.

- A$READ

  This system call applies to both data and directory files. Your application can use this system call to read file data from the location indicated by the file pointer. Before using this system call, your application can use the A$SEEK system call to position the file pointer. The A$READ 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 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 AGETFILE$STATUS system call in Chapter 8. The remaining 14 bytes are the ASCII characters making up the name of the file to which the directory entry points. (A file's name is the last component of a path name.) The advantage of using the A$READ system call to read a directory is that your application can obtain several entries with one operation.

- A$WRITE

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

- A$TRUNCATE

  This system call can be used only on data files. Your application can use this call to trim information from the end of the file. To do so, the application first must use A$SEEK to position the file pointer to the first byte to be dropped. Then the application invokes the A$TRUNCATE call to drop the specified byte and any bytes located after the specified byte. The A$TRUNCATE 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 AGETFILE$STATUS and A$GET$CONNECTION$STATUS. Both of these calls can be used with data files and directory files.

READING DIRECTORY ENTRIES

There are two system calls that your application can use to read entries from a directory. The A$READ system call (which can also be used to read a data file) was discussed earlier, under the heading "Manipulating Data." The second system call is AGETDIRECTORY$ENTRY. This system call can be used only on directory files, and can be used without opening a connection. The AGETDIRECTORY$ENTRY system call is fully described in Chapter 8 of this manual.

DELETING AND RENAMING FILES

The Basic 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:

- A$DELETE$FILE

  Your aplication 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.

  The process of deleting a file involves two stages. First, the application must call A$DELETE$FILE. This causes the file to be marked for deletion. The second stage, which is performed by the Basic I/O System, involves deciding when to delete the file. The Basic I/O System deletes marked files only after all connections to the file have been deleted. Refer to the A$DELETE$CONNECTION system call to see how to delete connections.

- A$RENAME$FILE

  Your application can use this system call to rename both data files and directory files. In renaming a file, your application 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 your application can change every component of a file's path name.

CHANGING ACCESS

The Basic I/O System provides one system call to let your application change a file's access list. This call is A$CHANGE$ACCESS, and it applies to both data files and directories. One rule governs the use of A$CHANGE$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.

ASCERTAINING A FILE'S NAME

The Basic I/O System provides a system call to let your application find out the last component of a file's path name when the application has a connection to the file. The system call is AGETPATH$COMPONENT, and you can use it on data files and directories. Note that your application can use this system call repeatedly to obtain the entire path name for a file.

MANIPULATING EXTENSION DATA

When you format a volume to accommodate named files, you have the option of allowing each file to carry extension data. The Basic I/O System provides two system calls that allow you to get and set extension data. These calls apply to both data and directory files, and both are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

- ASETEXTENSION$DATA

  This call provides a means of writing extension data. ASETEXTENSION$DATA can be used even if the file connection is not open.

- AGETEXTENSION$DATA

  This call provides a means of reading extension data. AGETEXTENSION$DATA can be used even if the file connection is not open.

DETECTING CHANGES IN DEVICE STATUS

The Basic I/O System provides the A$SPECIAL system call to allow your application to detect a change in the status of the device containing your named file tree. Specifically, your application can use the "notify" function of the A$SPECIAL system call to establish a mechanism for finding out if the device ceases to be ready. For more information, refer to the A$SPECIAL section of Chapter 8.

CHRONOLOGICAL OVERVIEW OF NAMED FILES

Although 31 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.

MOST FREQUENTLY USED SYSTEM CALLS

Figure 5-2 shows the chronological relationships between most frequently used Basic 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.

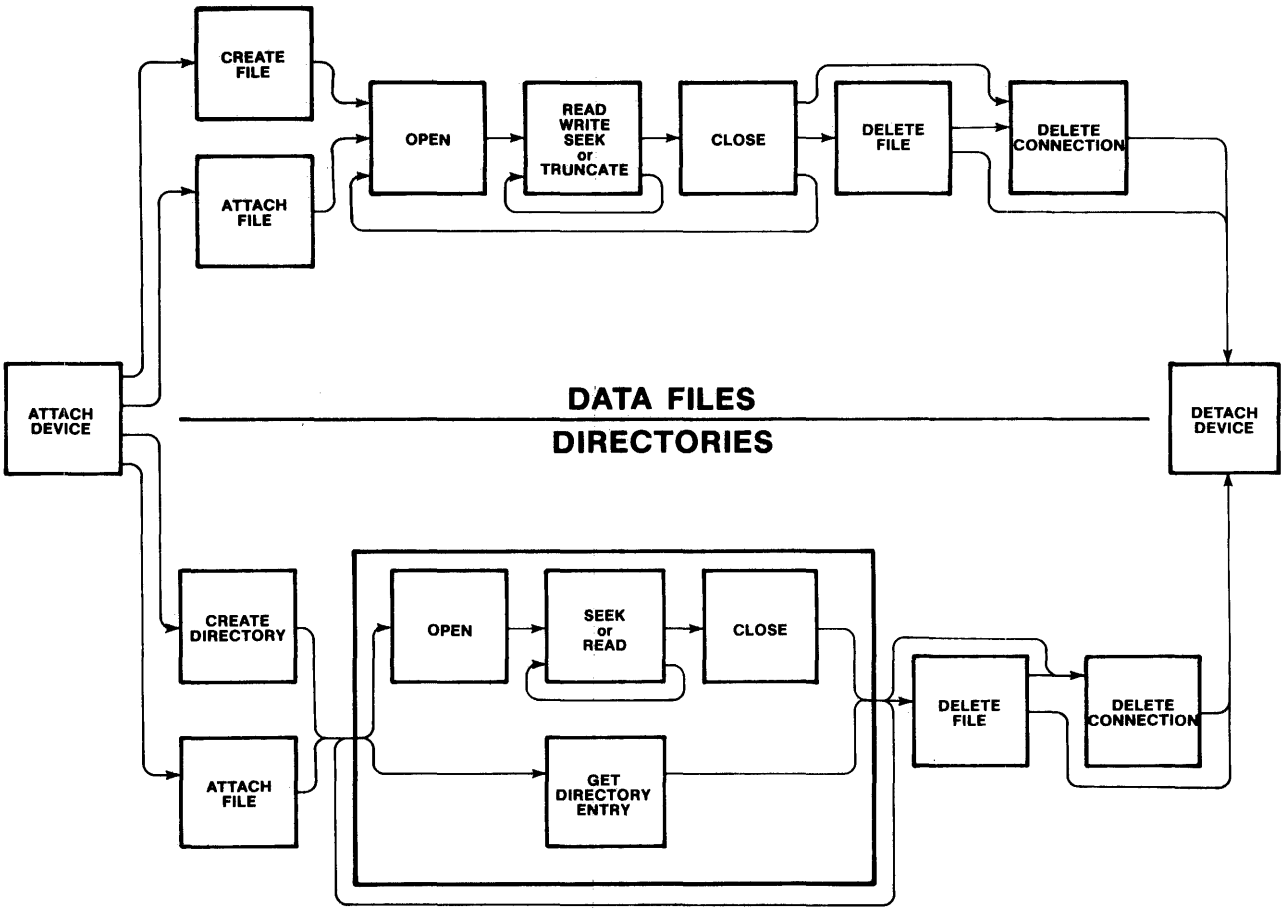

Figure 5-2. Chronology of Frequently Used System Calls for Named Files

CALLS RELATING TO USER OBJECTS

With one exception, the system calls relating to user objects are completely independent of other Basic 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$USER can be invoked any time. Two others, DELETE$USER and INSPECT$USER, can be invoked only after user objects exist. The remaining call, SET$DEFAULT$USER requires that both a job and a user object exist.

CALLS RELATING TO PREFIXES

The GET$DEFAULT$PREFIX system call can be invoked whenever a job exists. However, the SET$DEFAULT$PREFIX requires both a job and a user object.

CALLS RELATING TO STATUS

Both of the status-related system calls, AGETFILE$STATUS and A$GET$CONNECTION$STATUS, can be invoked whenever your application has a file connection.

CALLS RELATING TO CHANGING ACCESS

The only system call related to changing access, A$CHANGE$ACCESS, can be invoked whenever your application has both a user object and a path or connection to a file.

CALLS FOR MONITORING DEVICE READINESS

There is only one system call that lets your application monitor the readiness of a device, the A$SPECIAL system call. Your application can use the "notify" function of this call any time after your appication has obtained a device connection.

CALLS RELATING TO EXTENSION DATA

The two system calls relating to extension data, AGETEXTENSION$DATA and A$SET$EXTENSION$DATA, can be invoked whenever your application has a connection to a file.

CALLS FOR RENAMING FILES

The one call for renaming a file, A$RENAME$FILE, can be used whenever your application has a connection to the file to be renamed, a user object, and a path that is to become the new pathname.

CALLS FOR ASCERTAINING FILE NAMES

There is only one system call for finding out a file's name, AGETPATH$COMPONENT. Your application can use this call whenever the application has a connection to the file.

CHAPTER 6. PHYSICAL FILES

The Basic 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 Basic 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 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 to format a disk or diskette, the application must have access to every byte on the volume. Only physical files provide this kind of access.

- Volumes in Formats Required by Other Systems

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

- Implementing Your Own File Format

  Suppose that your application 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 Basic 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.

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 application creates the physical file, the device connection tells the Basic 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 A$PHYSICAL$ATTACH$DEVICE system call to obtain the device connection. When issuing this call, the system program must use the 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 A$PHYSICAL$ATTACH$DEVICE only once. The program can then save the device connection and pass it to any application program that requests it.

2. Obtain a file connection.

   If your application knows that the file has not yet been created, it should use the A$CREATE$FILE system call to obtain a file connection. This will work even though the physical file has already been created. When invoking the system call, set the USER, SUBPATH, ACCESS, MUST$CREATE, GRANULARITY, and SIZE parameters to zero, as these parameters are meaningless when creating a physical file. Use the token of the device connection as the PREFIX parameter in order to tell the Basic I/O System which device you want as your physical file.

   If, on the other hand, your application is certain that the file has already been created, use the A$ATTACH$FILE system call to obtain the file connection. To do this, your application should first obtain a connection to the device or an existing connection to the file and then use it as the PREFIX parameter in the system call. The application should set the USER and SUBPATH parameters to zero, as they have no meaning for physical files.

   This careful distinction between the A$CREATE$FILE and the A$ATTACH$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 A$OPEN system call to open the connection. When opening the connection, your application must specify how the file can be shared and how the application uses the connection. The system call chapter 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 A$READ and A$WRITE system calls are straightforward and are fully described in the system call chapter of this manual.

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

   - The A$SPECIAL system call can be used to request device dependent functions from the device driver. The precise nature of these functions depends upon the kind of device and the number of special functions supported by the device driver. Be aware that use of special functions can prevent an application from being device independent.

5. Close the file connection.

   Use the A$CLOSE system call to close the connection. This is particularly important if the share mode of the connection restricts the use of the file through other connections. Note that your application can repeat steps 2, 3, and 4 any number of times.

6. Delete the connection.

   Use the A$DELETE$CONNECTION system call to delete the connection. This is only necessary if your application is completely finished using the file.

7. Request that the device be detached.

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

CHAPTER 7. STREAM FILES

Stream files provide a means for one task to send large amounts of information to another task in a different job. Be aware that this is 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 are involved in using each stream file, each task must perform one half of a protocol. There are a large number of protocols that work, but the following one is typical and serves as a good illustration. Note that the two halves of the protocol can be performed in either order or concurrently.

ACTIONS REQUIRED OF THE WRITING TASK

The writing task must perform seven steps in its half of the protocol to ensure that it has established communication with the reading task. The steps are:

1. Obtain a connection to the stream file device.

   Although stream files do not actually require a physical device, your application must obtain a device connection before creating a stream file. This is necessary because, when your application invokes the A$CREATE$FILE system call, the device connection tells the Basic I/O System what kind of file to create.

   Since the process of attaching a device is restricted to system programs, you must create a system program that obtains the connection. This program must use the A$PHYSICAL$ATTACH$DEVICE system call to obtain the device connection.

   The A$PHYSICAL$ATTACH$DEVICE system call requires a parameter that identifies the device to be attached. For stream files, there is only one device, and its name is specified during the process of configuring the system. Intel recomends the name :stream:, but is is possible that the person responsible for configuring your system changed this name. For the remainder of this discussion, this manual assumes that the name of your system's stream file device is :stream:. For more information regarding the configuration process, refer the the iRMX 86 CONFIGURATION GUIDE.

As with other devices, :stream: cannot be multiply attached, so the system program should be written so as to call A$PHYSICAL$ATTACH$DEVICE only once. The program can then save the device connection and pass it to any application program that requests it.

2. Create the stream file.

   Use the A$CREATE$FILE system call to create the stream file. When invoking the system call, set the USER, SUBPATH, ACCESS, MUST$CREATE, SIZE, and GRANULARITY parameters to zero because these parameters have no meaning when creating a stream file. Use the token for the device connection as the PREFIX parameter in order to tell the Basic I/O System to create a stream file. If this system call runs successfully, the Basic I/O System will return a token for a file connection to the stream file.

3. Pass the file connection to the reading task.

   There are a number of ways of doing this, including object directories and mailboxes. For explicit instructions, refer to the iRMX 86 PROGRAMMING TECHNIQUES manual.

4. Open the file for writing.

   Use the A$OPEN system call to open the file connection for writing. Set the CONNECTION parameter to the token for the file connection. Set the MODE parameter to write. And set the SHARE parameter to allow sharing only with readers.

5. Write information to the stream file.

   Use the A$WRITE system call as often as needed to write information to the stream file. Use the token for the file connection as the CONNECTION parameter.

   The Basic I/O System uses the concurrent part of the A$WRITE system call to synchronize the writing and reading tasks on a call-by-call basis. The Basic I/O System does this by sending a response to each invocation of A$WRITE only after the reading task has finished reading all information that was written by the A$WRITE call.

6. Close the connection.

   When finished writing to the stream file, use the A$CLOSE system call to close the connection. Note that after this step, the writing task can repeat steps 4, 5, and 6.

7. Delete the connection.

   Use the A$DELETE$CONNECTION system call to delete the connection to the stream file.

ACTIONS REQUIRED OF THE READING TASK

The reading task must perform the following six steps in its half of the protocol to successfully read the information written by the writing task.

1. Get the file connection for the stream file.

   The technique used to accomplish this depends on how the writing task passed the file connection.

2. Create a second file connection for the stream file.

   There are two reasons for doing this. First, the reading task must have a different file pointer than the writing task. Second, the Basic I/O System rejects any connections created in one job but used by another to manipulate a file.

   Obtain this new connection by using the A$ATTACH$FILE system call. Set the USER and SUBPATH parameters to zero, and set the PREFIX parameter to the token for the original file connection.

   NOTE

   The reading task can also use the A$CREATE$FILE system call to obtain the new connection to the same stream file. The reason for this is that the Basic I/O System examines the nature of the PREFIX parameter in the A$CREATE$FILE system call. If the value provided is a device connection, the Basic I/O System will create a new file and return a connection for it. On the other hand, if the value provided is a file connection, the Basic I/O System will just create another connection to the same file.

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

3. Open the new file connection for reading.

   Use the A$OPEN system call to open the connection for reading. Set the CONNECTION parameter to the token for the new connection. Set the MODE parameter to read, and set the SHARE parameter to allow sharing with all connections to the file.

4. Commence reading.

   Use the A$READ system call to read the file until reading is no longer necessary or until an end-of-file condition is detected by the Basic I/O System.

5. Close the new file connection.

   Use the A$CLOSE system call to close the new file connection. Note that after this step, the reading task can repeat steps 3, 4, and 5.

6. Delete the new file connection.

   Use the A$DELETE$CONNECTION system call to delete the new connection to the stream file. The old connection is deleted by the writing task, and the stream file is deleted by the Basic I/O System as soon as both connections have been deleted.

CHAPTER 8. SYSTEM CALLS

This chapter describes the PL/M calling sequences to Basic I/O System calls. The system calls are listed here alphabetically by the same shorthand notation used throughout this manual. For example, A$DELETE$FILE refers to the asynchronous-level delete-file system call and appears alphabetically before SET$DEFAULT$PREFIX. This notation is language independent and should not be confused with the actual form of the PL/M call. The precise format of each call is spelled out as part of its detailed description.

For those I/O related calls which are invoked by system programmers, only the format of the call is described. Detailed descriptions of these calls are in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

Basic I/O operations are declared as typed or untyped external procedures for PL/M. PL/M programs perform I/O operations by issuing external procedure calls.

INPUT PARAMETER SPECIFICATION

The following paragraphs explain special properties of certain input parameters to Basic I/O System calls.

USER PARAMETER

This parameter is specified in some asynchronous system calls. It contains a token designating the caller's user object. A zero specification designates the default user. The Basic I/O System ignores this parameter for physical and stream files.

FILE-PATH PARAMETER(S) FOR NAMED FILES

Named files are designated in system calls by specifying their path, that is, their prefix and subpath. The prefix parameter can be a token designating an existing device connection or file connection. If this parameter is zero, the default prefix for the calling task's job is assumed.

For named files, the subpath parameter is a pointer to an ASCII string. The form of this string is described in the following paragraph. The subpath can also be zero or can point to a null string, in which case a prefix indicates the desired connection. For physical and stream files, the subpath parameter is always ignored.

System calls referring to named files can specify paths in the following forms:

Prefix	Subpath	Designated Connection
0	0 or a pointer to a null string	Connection whose token is the default prefix.
0	Pointer to ASCII string	ASCII string defines a path from the connection whose token is the default prefix to the target connection.
token	0 or a pointer to a null string	Connection whose token is contained in the prefix connection.
token	Pointer to ASCII string	Prefix parameter contains a token for a connection. ASCII string defines a path from that connection to the target connection.

The subpath ASCII string is a list of file names separated by slashes, terminating with the desired file. A file name can be 1-14 ASCII characters, including any printable ASCII character except the slash (/) and up-arrow (↑) or circumflex (^). In Figure 8-1, for example, if the prefix is the token for directory OBSTETRICS and we wish to reference file OUT_PATIENT, the subpath parameter must point to the string

DELIVERY/POST_PARTUM/OUT_PATIENT

If the ASCII string begins with a slash, the prefix merely designates the tree and the subpath is assumed to start at the root directory of the tree associated with the prefix. For example, if the prefix designates directory GYNECOLOGY in Figure 8-1, the subpath to OUT_PATIENT is

/OBSTETRICS/DELIVERY/POST_PARTUM/OUT_PATIENT

Named files can also be addressed relative to other files in the tree, using "↑" as a path component. The "↑" refers to the parent directory of the current file in the path scan. For example, now that we have a connection to OUT_PATIENT in Figure 8-1, we can use that connection to specify a subpath to IN_PATIENT. With the token for the OUT_PATIENT connection as our prefix, the subpath string would be

↑IN_PATIENT

Note that no slash follows the "↑" in this example.

Of course an even simpler approach would be to designate directory POST_PARTUM as the prefix, in which case the ASCII string becomes:

IN_PATIENT

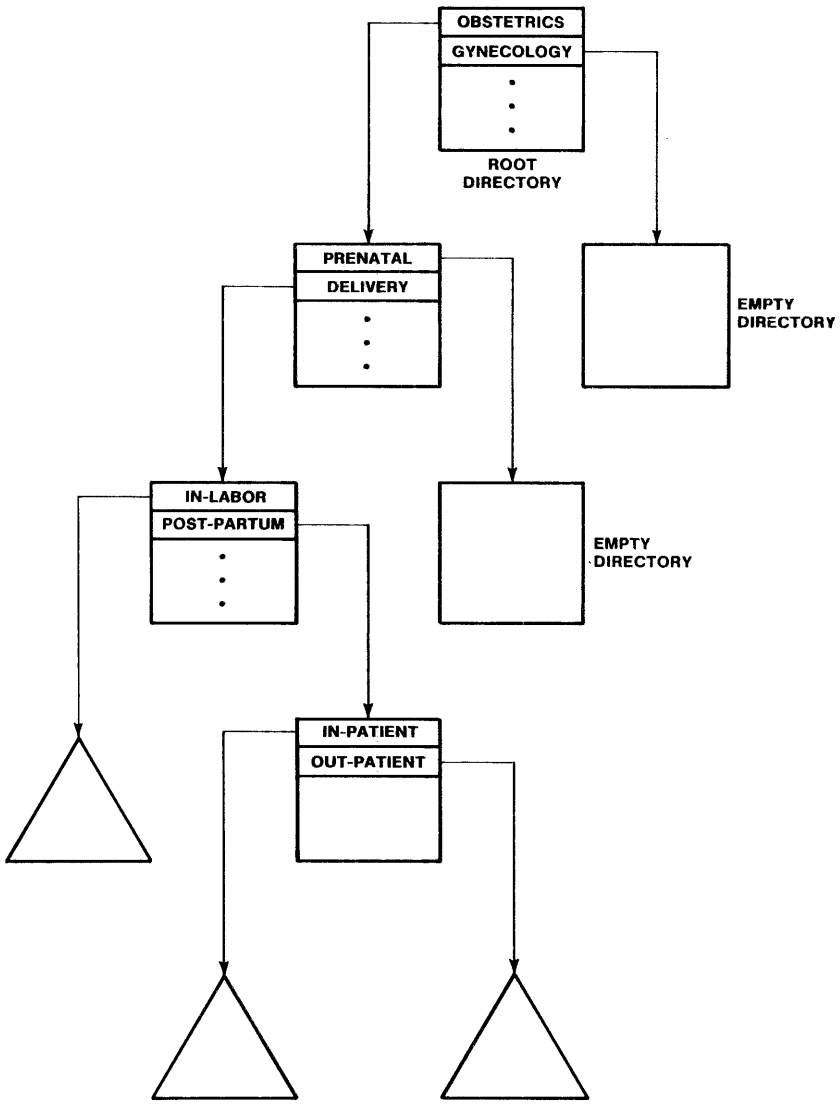

Figure 8-1. Sample Named File Tree

RESPONSE MAILBOX PARAMETER

This parameter is specified only in asynchronous system calls. It contains a token designating the mailbox that is to receive the result of the call. This information is provided by tasks to synchronize parallel operations. To receive the result of the call, a task must wait at the designated mailbox. Be aware that if several calls share the same mailbox, the results may be received out of order.

Most asynchronous system calls return only an I/O result segment to the response mailbox. This segment contains an exception code and other information about the operation. Appendix C describes the I/O result segment. Other system calls, the create-file-connection system calls, return to the mailbox a token for a connection if the system call completes sucessfully or an I/O result segment otherwise. After making one of these system calls, a task should perform a GET$TYPE system call to determine the type of object returned to the response mailbox. The iRMX 86 NUCLEUS REFERENCE MANUAL describes the GET$TYPE system call in detail.

NOTE

Result information segments should be deleted once they are no longer needed. Otherwise, they will consume available memory.

I/O BUFFERS

The A$READ and A$WRITE system calls each require a buffer while performing I/O. When you create these buffers, bear in mind the following restrictions:

- Once the I/O operation has been invoked, the tasks of your application should avoid changing the contents of the buffer until the Basic I/O System completes the operation.
- If you use an iRMX 86 segment as a buffer, be sure that the buffer is not deleted while an I/O operation is in progress.
- If you choose to use an iRMX 86 segment as a buffer, you should ensure that the segment is in the same job as the task performing the I/O operation. Using segments from one job as buffers for I/O operations in a different job can lead to a problem. For instance, suppose that Job A owns an iRMX 86 segment, and that Job B uses this segment as a buffer for I/O. If Job A is deleted, the iRMX 86 Operating System automatically deletes the buffer even if I/O is in progress.

EXCEPTION CODES

The Basic I/O System returns an exception code when a system call is invoked. If the call executes without error, the Basic I/O System returns the code "E$OK." If an error is encountered, some other code is returned.

For those system calls that do not require a response mailbox parameter, the Basic I/O System returns the exception code to the word pointed to by the except$ptr parameter. If an exceptional condition occurs, the Basic I/O System can then either return control to the calling task or pass control to an exception handler. See the iRMX 86 NUCLEUS REFERENCE MANUAL for a detailed description of exception handling.

For those system calls that do require a response mailbox parameter (the asynchronous calls), the Basic I/O System returns an exception code for the sequential portion of the call to the word pointed to by the except$ptr parameter and an exception code for the concurrent portion of the call to the status field of the I/O result segment (see Appendix C). If a sequential exceptional condition occurs, the Basic I/O System either returns control to the calling task or passes control to an exception handler. It does not process the asynchronous portion of the call. If a concurrent exceptional condition occurs, the calling task must signal the exception handler or process the exceptional condition in line.

SYSTEM CALLS

The following pages provide a detailed description of each Basic I/O System call, listed alphabetically. The system call dictionary, which appears first, provides a summary of these calls, grouped by function and correlated to the file types to which they apply. That system call dictionary also acts as a cross-reference to the detailed descriptions.

SYSTEM CALL DICTIONARY

This section summarizes the Basic I/O System calls by function and, where applicable, indicates the file types to which they apply:

PF	Physical file
SF	Stream file
NF	Named data file
ND	Named directory file

The page reference listed with each call points to the detailed description for the call.

JOB-LEVEL SYSTEM CALLS

System Call	Function	Page
SET$DEFAULT$PREFIX	Set default prefix for job.	8-105
GET$DEFAULT$PREFIX	Inspect default prefix.	8-99
SET$DEFAULT$USER	Set default user for job.	8-107
GET$DEFAULT$USER	Inspect default user.	8-101

GET TIME/DATE SYSTEM CALLS

System Call	Function	Page
GET$TIME	Get date/time value in internally-stored format.	8-103

CREATE-FILE-CONNECTION SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$CREATE$FILE	Asynchronous data file creation.	*	*	*		8-29
A$ATTACH$FILE	Asynchronous attach file.	*	*	*	*	8-9
A$CREATE$DIRECTORY	Asynchronous create directory.				*	8-23

FILE MODIFICATION SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$CHANGE$ACCESS	Asynchronous change access rights to file.			*	*	8-14
A$RENAME$FILE	Asynchronous rename file.			*	*	8-73

FILE INPUT/OUTPUT SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$OPEN	Asynchronous open file.	*	*	*	*	8-63
A$SEEK	Asynchronous move file pointer.	*		*	*	8-79
A$READ	Asynchronous read file.	*	*	*	*	8-69
A$WRITE	Asynchronous write file.	*	*	*		8-93
A$CLOSE	Asynchronous close file.	*	*	*	*	8-20

DEVICE-LEVEL FUNCTION SYSTEM CALL

System Call	Function	P F	S F	N F	N D	Page
A$SPECIAL	Asynchronous perform device-level function.	*	*	*		8-83

GET STATUS/ATTRIBUTE SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
AGETCON-NECTION$STATUS	Asynchronous get connection status.	*	*	*	*	8-44
AGETFILE$STATUS	Asynchronous get file status.	*	*	*	*	8-53
AGETDIRECTORY$ENTRY	Asynchronous inspect directory entry.				*	8-48
AGETPATH$COMPONENT	Asynchronous obtain path name from connection token.			*	*	8-60

DELETE CONNECTION/FILE SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$DELETE$CONNECTION	Asynchronous delete file connection.	*	*	*	*	8-36
A$TRUNCATE	Asynchronous truncate file.			*		8-90
A$DELETE$FILE	Asynchronous delete file.		*	*	*	8-39

SYSTEM PROGRAMMER CALLS (Calling Sequences Only)

System Call	Page
AGETEXTENSION$DATA	8-52
A$PHYSICAL$ATTACH$DEVICE	8-67
A$PHYSCIAL$DETACH$DEVICE	8-68
ASETEXTENSION$DATA	8-82
CREATE$USER	8-97
DELETE$USER	8-98
INSPECT$USER	8-104
SET$TIME	8-109

A$ATTACH$FILE

A$ATTACH$FILE creates a connection to an existing file.

```
CALL RQ$A$ATTACH$FILE(user, prefix, subpath, resp$mbox, except$ptr);
```

INPUT PARAMETERS

user	a WORD containing a token for the user object to be inspected in any access checking that takes place; a zero specifies the default user for the calling task's job; this parameter is ignored when attaching physical or stream files; access checking does occur for named files.
prefix	a WORD containing a token for the connection object to be used as the path prefix; normally, this will be a connection to an existing file (followed by a null subpath); a zero specifies the default prefix for the calling task's job.
subpath	a POINTER to a string containing the subpath of the file to be attached; a null string indicates that the new connection is to the file designated by the prefix; the new connection will not be open, regardless of the open state of the prefix.

OUTPUT PARAMETERS

resp$mbox	a WORD containing a token for the mailbox that receives the result object of the call; this result object is a new connection if the call succeeds or an I/O result segment otherwise (see Appendix C). To determine the type of object returned, use the Nucleus system call RQGETTYPE. If the object received is an I/O result segment, the calling task should issue RQ$DELETE$SEGMENT to delete the segment.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

A$ATTACH$FILE creates a connection to an existing file. Once the connection is established, it remains in effect until the connection object is deleted, or until the creating job is deleted. Once attached, the file may be opened, closed, read, written, etc., as many times as desired.

EXCEPTION CODES

A$ATTACH$FILE can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK — NORMAL CODE. No exceptional conditions.

EDEVOFFLINE — The prefix parameter in this system call refers to a logical connection. Either:

- The device is offline, or
- The device has never been physically attached. (See Appendix E for a more detailed explanation.)

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The prefix parameter
   - The response mailbox parameter
   - The user parameter.
2. The prefix connection is being deleted.

EXCEPTION CODES (continued)

E$LIMIT	Processing this call caused one of these limits to be exceeded: • The maximum number of objects allowed for this job (specified when the job was created). • The number of I/O operations which can be outstanding at one time for the user object specified in the call (255 decimal). • The number of I/O operations which can be outstanding at one time for the caller's job (also 255 decimal).
E$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	You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following: • When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default prefix. • No default prefix is cataloged for this job. • When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.
ENOUSER	If the user parameter in this call is not zero, then the problem is that the parameter is not a user object. If the user parameter is zero, it specifies a default user. But no default user can be found because: • When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default user. • No default user is cataloged for this job. • When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included. • The object which is cataloged with the name R?USER is not a user object. The name R?USER should be treated as a reserved word.

EXCEPTION CODES (continued)

ENOTCONFIGURED — One or more of the following system calls was not included when the system was configured:

A$ATTACH$FILE
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)
CREATE$COMPOSITE (Nucleus)

E$PARAM — The path name specified contains invalid characters.

E$TYPE — One of two conditions caused this exception:

- The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.)
- The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$CONTEXT — The file specified is on a device which the system is detaching.

E$FNEXIST — This indicates one of the following circumstances:

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

E$FTYPE — The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last string, each string in a pathname must be a named directory.)

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

EXCEPTION CODES (continued)

E$LIMIT	To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)
E$MEM	The memory pool of the Basic I/O System does not currently have a block of memory large enough to allow this system call to run to completion.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with A$ATTACH$FILE is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

A$CHANGE$ACCESS

A$CHANGE$ACCESS changes the access rights to a named data or directory file.

```
CALL RQ$A$CHANGE$ACCESS(user, prefix, subpath, id,  access, resp$mbox,
                        except$ptr);
```

INPUT PARAMETERS

user — a WORD containing a token for the user object to be inspected in access checking; a value of zero specifies the default user for the calling task's job.

prefix — a WORD containing a token for the connection to be used as the path prefix; typically, this would be a connection to the file whose access is being changed (followed by a null subpath); a zero specifies the default prefix for the calling task's job.

subpath — a POINTER to the STRING giving the subpath from the prefix to the file whose access is to be changed; a null string indicates that the prefix itself designates the desired file; in this case, the user parameter is ignored, since access checking was already performed when the file was attached.

id — a WORD giving the ID number of the user whose access is to be changed; if this ID does not already exist in the ID-access list, it is added; this list may contain a total of three ID-access pairs.

access — a BYTE mask giving the new access rights for the ID; if a bit is set to one, the corresponding access is granted; for a named data file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved (set to 0)

INPUT PARAMETERS

access (continued)

For a named directory file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved (set to 0)

If zero is specified for the access parameter (that is, no access), the ID specified in the id parameter is deleted from the file's ID-access list.

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives an I/O result segment indicating completion of the access change (see Appendix C). A value of zero means that you do not want to receive an I/O result segment.

If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

A$CHANGE$ACCESS system call applies to named files only. It is called to change the access rights to a named data or directory file. Depending on the contents of the "id" and "access" parameters specified in the system call, users may be added to or deleted from the files's ID-access list, or the access privileges granted to a particular user may be changed.

NOTE

The caller must be the owner of the file or must have change entry access to the file's parent directory. If the owner is "WORLD", that is, 0FFFFH, then any task may change the access mask of the file.

EXCEPTION CODES

A$CHANGE$ACCESS can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK — NORMAL CODE. No exceptional conditions.

EDEVOFFLINE — The prefix parameter in this system call refers to a logical connection. Either:

- The device is offline, or
- The device has never been physically attached. (See Appendix E for a more detailed explanation.)

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The prefix parameter
   - The response mailbox parameter
   - The user parameter.
2. The prefix connection is being deleted.

E$IFDR — This system call applies only to named files, but the prefix and subpath parameters specify some other type of file.

E$LIMIT — Processing this call caused one of these limits to be exceeded:

- The maximum number of objects allowed for this job (specified when the job was created).

EXCEPTION CODES

E$LIMIT (continued)

- The number of I/O operations which can be outstanding at one time for the user object specified in the call (255 decimal).
- The number of I/O operations which can be outstanding at one time for the caller's job (also 255 decimal).

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 — You specified a default prefix (prefix parameter equals zero). But no default prefix can be found because of one of the following:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default prefix.
- No default prefix is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.

ENOUSER — If the user parameter in this call is not zero, then the problem is that the parameter is not a user object.

If the user parameter is zero, it specifies a default user. But no default user can be found because:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default user.
- No default user is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.
- The object which is cataloged with the name R?USER is not a user object. The name R?USER should be treated as a reserved word.

EXCEPTION CODES (continued)

ENOTCONFIGURED	One or more of the following system calls was not included when the system was configured: CHANGE$ACCESS GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus) DISABLE$DELETION (Nucleus) CREATE$COMPOSITE (Nucleus)
E$PARAM	The path name specified contains invalid characters.
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.) • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The file specified is on a device which the system is detaching.
E$FACCESS	The user object in the parameter list is not qualified for "change entry access" for the parent directory, and is not the owner of the file.
E$FNEXIST	This indicates one of the following circumstances: • Either a file in the specified path, or the target file itself, does not exist. • Either a file in the specified path, or the target file itself, is marked for deletion.

EXCEPTION CODES (continued)

E$FTYPE	The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last string, each string in a pathname must be a named directory.)
E$IO	An I/O error occurred during the operation.
E$LIMIT	To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)
E$MEM	The memory pool of the Basic I/O System does not currently have a block of memory large enough to allow this system call to run to completion.
E$SUPPORT	Either of two problems can generate this exception code: 1. Your system is configured incorrectly. The entry point associated with A$CHANGE$ACCESS is not included in the "I/O System part" of the named file driver table. The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information. 2. The call attempted to add another access ID to the list of access ID's. The access list already contains the limit of three such ID's.

A$CLOSE

A$CLOSE closes an open file connection.

```
CALL RQ$A$CLOSE(connection, resp$mbox, except$ptr);
```

INPUT PARAMETER

connection	a WORD containing a token for the file connection to be closed.

OUTPUT PARAMETERS

resp$mbox	a WORD containing a token for a mailbox that is to receive an I/O result segment indicating the result of the operation (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$CLOSE system call closes an open file connection. It is called between uses of a file. A file connection must also be closed if the user wishes to change the open mode or shared status of the connection. The Basic I/O System will not close the connection until all existing I/O requests for the connection have been satisfied, and the Basic I/O system will not send a response to the resp$mbox until the file is closed.

EXCEPTION CODES

A$CLOSE can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

EXCEPTION CODES (continued)

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK — NORMAL CODE. No exceptional conditions.

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The connection parameter
   - The response mailbox parameter
2. The connection is being deleted.

E$LIMIT — To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)

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 — One or more of the following system calls was not included when the system was configured:

A$CLOSE
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)

E$SUPPORT — The connection parameter specified is not valid in this system call because the connection was not created by this job.

E$TYPE — One of two conditions caused this exception:

- The connection parameter is not a valid object type. It must be a connection object.
- The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection you are trying to close is not open.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with A$CLOSE is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

A$CREATE$DIRECTORY

A$CREATE$DIRECTORY creates a directory file.

```
CALL RQ$A$CREATE$DIRECTORY(user, prefix, subpath, access, resp$mbox,
                           except$ptr);
```

INPUT PARAMETERS

user — a WORD containing a token for the user object of the new directory's owner; the user object is inspected to make sure the caller has proper access to the new directory's parent; a zero specifies the default user for the calling task's job.

prefix — a WORD containing a token for the connection to be used as the path prefix; a zero specifies the default prefix for the calling task's job.

subpath — a POINTER to a STRING containing the subpath of the directory to be created; the subpath string must not be null, and must point to an unused location in the directory tree.

access — a BYTE mask giving the owner's initial access rights to the directory: for each bit in the mask, a one grants access and a zero denies it; the possible bit settings are:

Bit	Meaning
0	Delete
1	Display
2	Add Entry
3	Change Entry
4-7	Reserved (set to 0)

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives the result object of this call; this result object is a directory file connection if the call succeeded or an I/O result segment otherwise (see Appendix C). To determine the type of object returned, use the Nucleus system call GET$TYPE (Nucleus).

OUTPUT PARAMETERS

resp$mbox (continued)
: If the object received is an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.

except$ptr
: a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$CREATE$DIRECTORY system call is applicable to named directory files only. When called, it creates a new directory file and returns a token for the new file connection. This system call cannot be used to create a connection to an existing directory.

NOTE

The caller must have add-entry access to the parent of the new directory.

EXCEPTION CODES

A$CREATE$DIRECTORY can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK
: NORMAL CODE. No exceptional conditions.

EDEVOFFLINE
: The prefix parameter in this system call refers to a logical connection. Either:

- The device is offline, or
- The device has never been physically attached. (See Appendix E for a more detailed explanation.)

EXCEPTION CODES (continued)

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The prefix parameter
   - The response mailbox parameter
   - The user parameter.
2. The prefix connection is being deleted.

E$IFDR — This system call applies only to named files, but the prefix and subpath parameters specify some other type of file.

E$LIMIT — Processing this call caused one of these limits to be exceeded:

- The maximum number of objects allowed for this job (specified when the job was created).
- The number of I/O operations which can be outstanding at one time for the user object specified in the call (255 decimal).
- The number of I/O operations which can be outstanding at one time for the caller's job (also 255 decimal).

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 — You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default prefix.
- No default prefix is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.

ENOUSER — If the user parameter in this call is not zero, then the problem is that the parameter is not a user object.

EXCEPTION CODES

ENOUSER (continued)

If the user parameter is zero, it specifies a default user. But no default user can be found because:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default user.
- No default user is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.
- The object which is cataloged with the name R?USER is not a user object. The name R?USER should be treated as a reserved word.

ENOTCONFIGURED — One or more of the following system calls was not included when the system was configured:

A$CREATE$DIRECTORY
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)
CREATE$COMPOSITE (Nucleus)

E$PARAM — The path name contains invalid characters.

E$TYPE — One of two conditions caused this exception:

- The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.)
- The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$CONTEXT — The file specified is on a device which the system is detaching.

EXCEPTION CODES (continued)

E$FACCESS	The user object in the parameter list is not qualified for "add-entry" access to the parent directory.
E$FEXIST	The file you are trying to create already exists.
E$FNEXIST	This indicates one of the following circumstances: • A file in the specified path does not exist. • A file in the specified path is marked for deletion.
E$FTYPE	The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last string, each string in a pathname must be a named directory.)
E$IO	An I/O error occurred during the operation.
E$LIMIT	To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)
E$MEM	The memory pool of the Basic I/O System does not currently have a block of memory large enough to allow this system call to run to completion.
E$SPACE	Either: • The volume has no more space, or • No more named files or directories can be created on this volume. The maximum number of files or directories which can be created on a particular volume is set when that volume is formatted. (See the description of the FORMAT Command in the iRMX 86 HUMAN INTERFACE REFERENCE MANUAL.)
E$SUPPORT	This code is caused by a conflict between the service being requested by the A$CREATE$DIRECTORY call and the way the Basic I/O System is configured. One of these conditions exists: • The entry point associated with A$CREATE$DIRECTORY is not included in the "I/O System part" of the named file driver table. The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

EXCEPTION CODES

E$SUPPORT (continued)

- The call is attempting to allocate file space, but the Basic I/O System was configured with an option which prevents allocation of file space.

A$CREATE$FILE

A$CREATE$FILE creates a physical, stream, or named file.

```
CALL RQ$A$CREATE$FILE(user, prefix, subpath, access, granularity,
                      high$size, low$size, must$create, resp$mbox,
                      except$ptr);
```

INPUT PARAMETERS

user	applies to named files only and is a WORD containing a token for the user object of the file's owner; it also furnishes the user ID for any access checking that might occur; a zero specifies the default user for the job; this parameter is ignored for physical and stream files.
prefix	a WORD containing a token for a device or file connection; by implication, this parameter indicates the type of file (physical, stream, named) being created; for stream files, if the prefix is a device connection, a new stream file is created, and if the prefix is a file connection, a new file connection to the same stream file is created; for named files, the prefix acts as the starting point in a directory tree scan; a zero specifies the default prefix for the job.
subpath	applies to named files only and is a POINTER to a STRING containing the subpath for the file being created.
access	applies to named files only and is a BYTE mask giving the owner's initial access rights to the new file; for each bit, a one grants access and a zero denies it; the possible bit settings are:

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved (set to 0)

INPUT PARAMETERS (continued)

granularity

applies to named files only and is a WORD giving the granularity of the file being created; this is the size (in bytes) of each logical block to be allocated to the file; the value specified in this parameter is rounded up, if necessary, to a multiple of the volume granularity; note that a contiguous file can be expanded into a noncontiguous file by writing past the contiguous boundaries.

The granularity parameter can have the following values:

0	Same as volume granularity
0FFFFH	The file must be contiguous
other	Number of bytes/allocation

When a contiguous file is extended, space is allocated in volume-granularity units; if "other" is specified, a multiple of 1024 bytes is recommended.

This parameter is ignored for physical and stream files.

high$size
low$size

applies to named files only and is a WORD pair giving the number of bytes initially reserved for the file; for stream files, this value must equal zero.

must$create

applies to named files only and is a BYTE whose value (0FFH for true or 0 for false) determines the handling of input paths designating an existing file (see following DESCRIPTION).

OUTPUT PARAMETERS

resp$mbox

a WORD containing a token for the mailbox that receives the result object of this call; this result object is a token for a new file connection if the call succeeded or a token for an I/O result segment otherwise (see Appendix C). To determine the type of object returned, use the Nucleus system call GET$TYPE (Nucleus).

If the object received is an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.

OUTPUT PARAMETERS (continued)

except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$CREATE$FILE system call creates a physical, stream, or named data file and returns a token for the new file connection. If a named file designated by the prefix and subpath parameters already exists, one of the following situations occurs:

- Error: If the "must$create" parameter is TRUE (0FFH), an error condition code (E$FEXIST) is returned.

- Truncate File: If the "must$create" parameter is FALSE (0) and the path designates an existing data file, a new connection to that file is returned (that is, A$CREATE$FILE acts like A$ATTACH$FILE). In this case, the file is truncated or expanded according to the "size" parameter, so data in the file might be lost.

- Temporary File Created: If the "must$create" parameter is FALSE (0), and the path designates an existing directory file or device, an unnamed temporary file is created on the corresponding device. This file is deleted automatically when the last connection to it is deleted. Since this file is created without a path, it can be accessed only through a connection.

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

Many of the parameters specified in the A$CREATE$FILE call do not apply to physical and stream files. In these cases, the parameter is ignored.

NOTE

The caller must have add-entry access to the parent directory of the new named file.

EXCEPTION CODES

A$CREATE$FILE can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK — NORMAL CODE. No exceptional conditions.

EDEVOFFLINE — The prefix parameter in this system call refers to a logical connection. Either:

- The device is offline, or
- The device has never been physically attached. (See Appendix E for a more detailed explanation.)

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The prefix parameter
   - The response mailbox parameter
   - The user parameter.
2. The prefix connection is being deleted.

E$LIMIT — To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)

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 — You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default prefix.
- No default prefix is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.

ENOUSER — If the user parameter in this call is not zero, then the problem is that the parameter is not a user object.

If the user parameter is zero, it specifies a default user. But no default user can be found because:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default user.
- No default user is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.
- The object which is cataloged with the name R?USER is not a user object. The name R?USER should be treated as a reserved word.

ENOTCONFIGURED — One or more of the following system calls was not included when the system was configured:

CREATE$FILE
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)
CREATE$COMPOSITE (Nucleus)

E$PARAM — The path name contains invalid characters.

E$TYPE — One of two conditions caused this exception:

- The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.)

EXCEPTION CODES

E$TYPE (continued)

- The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$CONTEXT	The file specified is on a device which the system is detaching.
E$FACCESS	The user object in the parameter list is not qualified for "add entry" to the parent directory, or is not qualified for "update" access to existing file.
E$FEXIST	The "must$create" parameter in the call is TRUE, and the file already exists. (See the DESCRIPTION section.)
E$FNEXIST	This indicates one of the following circumstances: • A file in the specified path does not exist. • A file in the specified path is marked for deletion.
E$FTYPE	The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last string, each string in a pathname must be a named directory.)
E$IO	An I/O error occurred during the operation.
E$LIMIT	To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)
E$MEM	The memory pool of the Basic I/O System does not currently have a block of memory large enough to allow this system call to run to completion.

EXCEPTION CODES (continued)

E$SHARE — The file this call is attempting to create already exists and is open. It was opened with the characteristic "no share with writers." (See the A$OPEN call in this chapter.)

E$SPACE — Either:

- The volume has no more space, or
- No more named files or directories can be created on this volume. The maximum number of files or directories which can be created on a particular volume is set when that volume is formatted. (See the description of the FORMAT Command in the iRMX 86 HUMAN INTERFACE REFERENCE MANUAL.)

E$SUPPORT — This code is caused by a conflict between the service being requested by the A$CREATE$FILE call and the way the Basic I/O System is configured. One of these conditions exists:

- The entry point associated with A$CREATE$FILE is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.
- The call is attempting to allocate file space, but the Basic I/O System was configured with an option which prevents allocation of file space.
- The file exists, and the must$create parameter is FALSE. When the Basic I/O System was configured, an option was chosen which prevented this combination, so that files could not be automatically truncated to zero size. See the DESCRIPTION section.
- The file exists, the size parameter in the call is less than the current size of the file, and the Basic I/O System was configured with an option which prevents truncation of files.

A$DELETE$CONNECTION

A$DELETE$CONNECTION deletes a named file connection created by A$CREATE$FILE, A$CREATE$DIRECTORY, or A$ATTACH$FILE.

```
CALL RQ$A$DELETE$CONNECTION(connection, resp$mbox, except$ptr);
```

INPUT PARAMETER

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

OUTPUT PARAMETERS

resp$mbox	a WORD containing a token for the mailbox that receives an I/O result segment indicating the result of the operation (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$DELETE$CONNECTION system call deletes a connection object. It also deletes the associated file if both the file is already marked for deletion (by a previous A$DELETE$FILE call) and the specified connection is the last remaining connection to the file. If a connection is open when A$DELETE$CONNECTION is called, it is closed before being deleted.

NOTE

Connections should be deleted when no longer needed.

EXCEPTION CODES

A$DELETE$CONNECTION can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK — NORMAL CODE. No exceptional conditions.

E$CONTEXT — The connection parameter is a device connection, not a file connection.

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The connection parameter
   - The response mailbox parameter
2. The connection is being deleted.

E$LIMIT — To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)

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 — One or more of the following system calls was not included when the system was configured:

A$DELETE$CONNECTIION
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)

EXCEPTION CODES (continued)

E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.) • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$IO	An I/O error occurred during the operation.
E$SUPPORT	One of these conditions caused this exception code: • Your system is configured incorrectly. The entry point associated with A$DELETE$CONNECTION is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information. • The connection being deleted is the last connection to a file which is marked for deletion. Normally this would be the point at which the file itself would be deleted. But the Basic I/O System was configured without the capability to delete files. (The call to delete the file would also have returned an E$SUPPORT exception code.)

A$DELETE$FILE

A$DELETE$FILE marks a named or stream file for deletion.

```
CALL RQ$A$DELETE$FILE(user, prefix, subpath, resp$mbox, except$ptr);
```

INPUT PARAMETERS

user — applies to named files only and is a WORD containing a token for the user object to be inspected in access checking; a zero specifies the default user for the calling task's job.

prefix — a WORD containing a token for a connection; in the case of a named file, this prefix acts as the starting point in a directory tree scan; a zero specifies the default prefix for the calling task's job.

subpath — applies to named files only and is a POINTER to a STRING giving the subpath for the file being deleted; a null string indicates that the prefix itself designates the desired file; in this instance, the user parameter is ignored, since access checking was already performed when the file was attached.

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for a mailbox that receives an I/O result segment (see Appendix C) when the file is marked for deletion. The file will not actually be deleted until all connections to the file are deleted, as explained under the DESCRIPTION below. A value of zero means that you do not want to receive an I/O result segment.

If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion and removes the file's entry from the parent directory. The entry is removed immediately, but the file is not actually deleted until all connections to the file have been severed (by A$DELETE$CONNECTION calls). Directory files cannot be deleted unless they are empty.

NOTE

The caller must have delete access to the file.

EXCEPTION CODES

A$DELETE$FILE can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK — NORMAL CODE. No exceptional conditions.

EDEVOFFLINE — The prefix parameter in this system call refers to a logical connection. Either:

- The device is offline, or
- The device has never been physically attached. (See Appendix E for a more detailed explanation.)

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The prefix parameter

EXCEPTION CODES

E$EXIST (continued)

- The response mailbox parameter
- The user parameter.

2. The prefix connection is being deleted.

E$IFDR — This system call applies only to named or stream files, but the parameter list specified a another type of file.

E$LIMIT — Processing this call caused one of these limits to be exceeded:

- The maximum number of objects allowed for this job (specified when the job was created).
- The number of I/O operations which can be outstanding at one time for the user object specified in the call (255 decimal).
- The number of I/O operations which can be outstanding at one time for the caller's job (also 255 decimal).

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 — You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default prefix.
- No default prefix is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.

ENOUSER — If the user parameter in this call is not zero, then the problem is that the parameter is not a user object.

If the user parameter is zero, it specifies a default user. But no default user can be found because:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default user.

EXCEPTION CODES

ENOUSER (continued)

- No default user is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.
- The object which is cataloged with the name R?USER is not a user object. The name R?USER should be treated as a reserved word.

ENOTCONFIGURED — One or more of the following system calls was not included when the system was configured:

DELETE$FILE
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)
CREATE$COMPOSITE (Nucleus)

E$PARAM — The subpath parameter contains invalid characters.

E$SUPPORT — The connection parameter specified is not valid in this system call because the connection was not created by this job.

E$TYPE — One of two conditions caused this exception:

- The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.)
- The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK — NORMAL CODE. No exceptional conditions.

E$CONTEXT — One of these problems caused this exception code:

- The file specified is on a device which the system is detaching.

EXCEPTION CODES

E$CONTEXT (continued)

- The call is attempting to delete a stream file which is already marked for deletion.
- The call is attempting to delete a directory which has entries in it, or is attempting to delete a ROOT directory.

E$FACCESS — The user object in the parameter list is not qualified for "delete" access to this file.

E$FNEXIST — This indicates one of the following circumstances:

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

E$FTYPE — The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last string, each string in a pathname must be a named directory.)

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

E$LIMIT — To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)

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

E$SUPPORT — One of two problems exist:

- Your system is configured incorrectly. The entry point associated with A$DELETE$FILE is not included in the "I/O System part" of the file driver table (named or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.
- The Basic I/O System was configured to prevent truncation of files. In order to delete a file, the file must first be truncated.

AGETCONNECTION$STATUS

AGETCONNECTION$STATUS returns information about a file connection.

```
CALL RQ$A$GET$CONNECTION$STATUS(connection, resp$mbox, except$ptr);
```

INPUT PARAMETER

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

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives a connection-status segment. The calling task is responsible for deleting the connection-status segment.

The information in this segment is structured as follows:

```
DECLARE
  conn$status     STRUCTURE(
    status             WORD,
    file$driver        BYTE,
    flags              BYTE,
    open$mode          BYTE,
    share              BYTE,
    low$file$ptr       WORD,
    high$file$ptr      WORD,
    access             BYTE);
```

These fields are interpreted as follows:

status — a condition code indicating how the status-fetch operation completed; if this code is not E$OK, the remaining fields must be considered invalid.

file$driver — tells the type of file driver to which this connection is attached; possible values are:

Value	Type
1	Physical files
2	Stream files
4	Named files

OUTPUT PARAMETERS

resp$mbox (continued)

flags — contains two flag bits; if bit 1 is set to one, this connection is active and can be opened; if bit 2 is set, this connection is a device connection.

open$mode — the mode established when this connection was opened; possible values are:

0	Connection is closed
1	Open for reading
2	Open for writing
3	Open for reading and writing

share — the current sharing status established when this connection was opened; possible values are:

0	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

The open mode and shared state are initially set by the A$OPEN call.

low$file$ptr
high$file$ptr — the current byte location of the file pointer for this connection.

access — gives the access rights for this connection; for each bit set to one, the corresponding access is granted; bit values are:

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

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The AGETCONNECTION$STATUS system call returns a segment containing status information about a file connection.

EXCEPTION CODES

AGETCONNECTION$STATUS can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter 2. The connection is being deleted.
E$LIMIT	To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)
E$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	One or more of the following system calls was not included when the system was configured: AGETCONNECTION$STATUS GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.

EXCEPTION CODES (continued)

E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System will return the following codes in the connection-status segment (status field). After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with AGETCONNECTION$STATUS is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

AGETDIRECTORY$ENTRY

AGETDIRECTORY$ENTRY returns the file name associated with a named directory file entry.

```
CALL RQ$A$GET$DIRECTORY$ENTRY(connection, entry$num, resp$mbox,
                              except$ptr);
```

INPUT PARAMETERS

connection — a WORD containing a token for the directory file with the desired entry.

entry$num — a WORD giving the entry number of the desired file name; entries within a directory are numbered sequentially starting from zero; E$EMPTY$ENTRY condition code will be issued if the specified file has been deleted and the Basic I/O System has not reissued the entry to another file.

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives a directory-entry segment. The task making the AGETDIRECTORY$ENTRY call is responsible for deleting this segment.

Information in this segment is structured as follows:

```
DECLARE
  dir$entry$info  STRUCTURE(
     status         WORD,
     name (14)      BYTE);
```

where:

status — indicates how the operation completed; EOK, EEMPTY$ENTRY, and E$DIR$END condition codes all indicate successful completion.

name — the file name contained in the designated entry, padded with blanks; this field is valid only if status = E$OK.

OUTPUT PARAMETERS (continued)

except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The AGETDIRECTORY$ENTRY system call applies to named files only. When called, it returns the file name associated with a specified directory entry. This name is a single subpath component for a file whose parent is the designated directory. As an alternative to using this system call, an application task can open and read a directory file.

NOTE

The caller must have display access to the designated directory.

EXCEPTION CODES

AGETDIRECTORY$ENTRY can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter 2. The connection is being deleted.

EXCEPTION CODES (continued)

E$IFDR	This system call applies only to named directories, but the connection parameter specifies something else.
E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	One or more of the following system calls was not included when the system was configured: AGET$DIRECTORY$ENTRY GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System will return the following codes in the directory-entry segment (status field). After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$DIREND	The entry$num parameter is greater than the number of entries in the directory.
E$EMPTY$ENTRY	The file entry designated in the call has been deleted, and the Basic I/O System has not reissued the entry to another file.
E$FACCESS	The connection in the parameter list is not qualified for "display" access to the directory.
E$FTYPE	The connection parameter does not refer to a directory.

EXCEPTION CODES (continued)

E$IO	An I/O error occurred during the operation.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with GET$DIRECTORY$ENTRY is not included in the "I/O System part" of the named file driver table. The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

AGETEXTENSION$DATA

The AGETEXTENSION$DATA system call returns extension data stored with a Basic I/O System file.

```
CALL RQ$A$GET$EXTENSION$DATA(connection, resp$mbox, except$ptr);
```

This System Programmer call is included here for convenience. AGETEXTENSION$DATA is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

AGETFILE$STATUS

AGETFILE$STATUS returns status and attribute information about a file.

```
CALL RQ$A$GET$FILE$STATUS(connection, resp$mbox, except$ptr);
```

INPUT PARAMETER

connection — a WORD containing a token for a connection to the file whose status is sought.

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives a segment containing the common file-status (and, for named files, the named file-status) information. The calling task is responsible for deleting this segment.

Structure of the common file-status information is as follows:

```
DECLARE  common$info  STRUCTURE(
    status          WORD,
    num$conn        WORD,
    num$reader      WORD,
    num$writer      WORD,
    open$share      BYTE,
    named$file      BYTE,
    dev$name (14)   BYTE,
    file$drivers    WORD,
    functs          BYTE,
    flags           BYTE,
    dev$gran        WORD,
    low$dev$size    WORD,
    high$dev$size   WORD,
    dev$conn        WORD);
```

These fields are interpreted as follows:

status — a condition code indicating how the status-fetch operation completed; if this code is not E$OK, the remaining fields must be considered invalid.

num$conn — the number of connections to the file.

OUTPUT PARAMETERS

resp$mbox (continued)

num$reader — the number of connections currently open for reading.

num$writer — the number of connections currently open for writing.

open$share — the current shared status of the file; possible values are:

0	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

named$file — specifies whether the file is a named file and, therefore, whether the segment will contain named-file, as well as common information; a value of OFFH indicates the additional information is present.

dev$name — the name of the device where this file resides, padded with blanks.

file$drivers — indicates which file drivers can be used with the device containing the file; if bit n is on, then file driver n+1 can be used; bit 15 is the high-order bit.

Bit	Driver No.	Driver
0	1	Physical file
1	2	Stream file
2	3	reserved
3	4	Named file

functs — describes the functions supported by the device where this file resides; each bit set to one indicates the corresponding function is supported.

Bit	Function
0	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE

OUTPUT PARAMETERS

resp$mbox (continued)

flags — used only with diskette drives; interpreted as follows:

Bit	Meaning
0	not a diskette device
1	0=single density 1=double density
2	0=single sided 1=double sided
3	0=8-inch diskette 1=5 1/4-inch diskette
4-7	reserved

where bit 0 is the rightmost bit.

dev$gran — the device granularity, in bytes.

lowdevsize
highdevsize — the size of the device, in bytes.

dev$conn — the number of connections to the device.

The foregoing structure is returned for all files. If the file is a named file, additional information is returned. This information appears in the segment immediately after the common file-status information (described previously) and is structured as follows:

```
DECLARE
   named$file$info  STRUCTURE(
   fdesc$num         WORD,
   file$type         BYTE,
   file$gran         BYTE,
   owner             WORD,
   low$create$time   WORD,
   high$create$time  WORD,
   low$access$time   WORD,
   high$access$time  WORD,
   low$mod$time      WORD,
   high$mod$time     WORD,
   low$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,
```

OUTPUT PARAMETERS

resp$mbox (continued)

```
                    id$count            WORD,
                    first$access        BYTE,
                    first$ID            WORD,
                    second$access       BYTE,
                    second$ID           WORD,
                    third$access        BYTE,
                    third$ID            WORD);
```

These fields are interpreted as follows:

fdesc$num	the number of the file's file descriptor; the file descriptor is a Basic I/O System data structure containing file attribute and status data.
file$type	indicates the type of the file; a value of 8 means data file, and 6 means directory file.
file$gran	specifies the file granularity.
owner	the user ID number of the file's owner.
low$create$time high$create$time	the time and date when the file was created; whether the Basic I/O System maintains this field depends on a configuration option.
low$access$time high$access$time	the time and date when the file was last accessed; whether the Basic I/O System maintains this field depends on a configuration option.
lowmodtime highmodtime	the time and date when the file was last modified; whether the Basic I/O System maintains this field depends on a configuration option.
low$file$size high$file$size	the total size, in bytes, of the data in the file.
low$file$blocks high$file$blocks	the number of volume blocks allocated to this file.
vol$name	the ASCII name for the volume containing this file.

OUTPUT PARAMETERS

resp$mbox (continued)

vol$gran	the volume granularity, in bytes.
lowvolsize highvolsize	the size of the volume, in bytes.
id$count	the number of access/ user-ID pairs declared for this file.
first$access second$access third$access	access masks for as many ID's as are indicated by id$count.

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

first$ID second$ID third$ID	ID values for the accessors.

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The AGETFILE$STATUS system call returns status and attribute information about the designated file. Certain common information is returned regardless of the file driver type (physical, stream, or named). Additional information is returned for named files.

Note that this call returns device-dependent information.

EXCEPTION CODES

AGETFILE$STATUS can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter 2. The connection is being deleted.
E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).
E$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	One or more of the following system calls was not included when the system was configured: AGETFILE$STATUS GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System will return the following codes in the status segment (staus field) at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

EXCEPTION CODES (continued)

E$OK	NORMAL CODE. No exceptional conditions.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with AGETFILE$STATUS is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

AGETPATH$COMPONENT

AGETPATH$COMPONENT returns the name of a named file as the file is known in its parent directory.

```
CALL RQ$A$GET$PATH$COMPONENT(connection, resp$mbox, except$ptr);
```

INPUT PARAMETER

connection — a WORD containing a token for the file connection whose name is sought.

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives the file$name segment; this segment contains the file name associated with the designated connection and is structured as follows:

```
DECLARE FILE$NAME
  file$name  STRUCTURE(
    status     WORD,
    name(14)   BYTE);
```

These fields are interpreted as follows:

where:

- **status** — a condition code indicating how the operation completed.
- **name** — a left-justified, null-padded ASCII string giving the desired file name; this name is the same as the last item in the subpath string specified when the file was created or renamed.

NOTE

The task which makes the AGETPATH$COMPONENT call is responsible for deleting the file$name segment.

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

A caller who knows the token for a connection to a file can specify the token to this system call and receive the name of the file in return. This is the name by which the file is cataloged in its parent directory. If the connection is to the root directory of a volume (that is, if no parent directory exists), a null string is returned. A null string is also returned if the file is marked for deletion.

AGETPATH$COMPONENT can be called no matter what type of file is supported, but if a connection to a physical or stream file is specified, the call simply returns a null string.

EXCEPTION CODES

AGETPATH$COMPONENT can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK — NORMAL CODE. No exceptional conditions.

E$EXIST — Two conditions can cause this exception code to be returned:

1. At least one of the following parameters is not a token for a valid object:
   - The connection parameter
   - The response mailbox parameter
2. The connection is being deleted.

E$LIMIT — The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).

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.

EXCEPTION CODES (continued)

ENOTCONFIGURED	One or more of the following system calls was not included when the system was configured: AGETPATH$COMPONENT GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System will return the following codes in the status field of the file$name segment. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$FNEXIST	The file is marked for deletion. (A null string is returned in the name field of the file$name segment.)
E$IO	An I/O error occurred during the operation.
E$LIMIT	To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)
E$MEM	The memory pool of the Basic I/O System does not currently have a block of memory large enough to allow this system call to run to completion.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with AGETPATH$COMPONENT is not included in the "I/O System part" of the named file driver table. The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

A$OPEN

A$OPEN opens an asynchronous file connection for I/O operations.

```
CALL RQ$A$OPEN(connection, mode, share, resp$mbox, except$ptr);
```

INPUT PARAMETERS

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

mode — a BYTE giving the mode desired for the open connection; possible values are:

1	Open for reading
2	Open for writing
3	Open for both reading and writing

share — a BYTE specifying the kind of sharing desired for this connection; possible values are:

0	Private use only
1	Share with readers only
2	Share with writers only
3	Share with all users

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives the I/O result segment (see Appendix C) indicating completion of this operation. A value of zero means that you do not want to receive an I/O result segment.

If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$OPEN system call opens a connection for I/O operations. The connection must be opened before reading, writing, and seeking can be performed on the associated file.

A$OPEN also initializes the file pointer to byte position zero. Subsequent Basic I/O System calls (A$SEEK, A$READ, and A$WRITE) will move this pointer.

A$OPEN checks the current sharing status of the file, and returns an E$SHARE exceptional condition if the requested sharing status is inconsistent with the sharing already permitted. Open requests are not queued.

If the file is attached by multiple connections, the file might be open for reading by some connections and open for writing by others at the same time. Any modification of the file by a writer will be seen by the reader, if a reader subsequently reads the modified part of the file.

The request mode is compared to the current sharing status of the file; if they are not compatible, an E$SHARE exceptional condition is returned. No deadlock occurs, however, since open calls are not queued. The system does not notify callers when the sharing status of the connection changes. If such notification is important, users of the file should arrange a suitable protocol.

NOTES

The mode and share parameters must each be compatible with the current shared state of the connected file.

Directory files can be opened and read, but only by specifying a one for the mode parameter and a three for the share parameter.

EXCEPTION CODES

A$OPEN can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter 2. The connection is being deleted.
E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).
E$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	One or more of the following system calls was not included when the system was configured: A$OPEN GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$PARAM	The mode or share parameter has an invalid value (out of the range 1-3 or 0-3 respectively).
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection (file or directory) is already open, or it is a device connection.
E$FACCESS	The connection does not have access compatible with the mode specified in this A$OPEN call.
E$SHARE	One of these situations prevented opening the file: • The current file share characteristic is not compatible with the mode or the share parameter in the A$OPEN call. • This A$OPEN is attempting to open a directory for some operation other than "read" (mode parameter) and "share with all users" (share parameter). (See DESCRIPTION above for more information on sharing of files.)
E$SUPPORT	Your system is configured incorrectly. The entry point associated with A$OPEN is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

A$PHYSICAL$ATTACH$DEVICE

The A$PHYSICAL$ATTACH$DEVICE system call attaches a device to the Basic I/O System.

```
CALL RQ$A$PHYSICAL$ATTACH$DEVICE(dev$name, file$driver, resp$mbox,
                                 except$ptr);
```

This System Programmer call is included here for convenience. A$PHYSICAL$ATTACH$DEVICE is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

A$PHYSICAL$DETACH$DEVICE

The A$PHYSICAL$DETACH$DEVICE system call detaches a device from the Basic I/O System.

```
CALL RQ$A$PHYSICAL$DETACH$DEVICE(connection, hard, resp$mbox,
                                 except$ptr);
```

This System Programmer call is included here for convenience. A$PHYSICAL$DETACH$DEVICE is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

A$READ

A$READ reads the requested number of bytes, starting with the current position of the pointer for the specified file.

```
CALL RQ$A$READ(connection, buff$ptr, count, resp$mbox, except$ptr);
```

INPUT PARAMETERS

connection	a WORD containing a token for the open file connection to be read.
buff$ptr	a POINTER to the buffer that receives the data.
count	a WORD giving the number of bytes to be read.

OUTPUT PARAMETERS

resp$mbox	a WORD containing a token for the mailbox that receives the I/O result segment (see Appendix C) after the read is complete. A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment. The number of bytes read is in the "actual" field of the I/O result segment. If a read operation is requested with the file pointer set at or beyond the end of the file, an actual value of of zero is returned. If all the connections to a stream file are requesting read operations, an actual value of zero is returned.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$READ system call initiates a read operation from an open connection. The connection is read as a string of bytes, starting at the current location of the file pointer. Any number of bytes can be requested. Some efficiency may be gained by starting reads on device block boundaries. After the read operation is finished, the file pointer points just past the last byte read.

The buffer specified by the "buff$ptr" parameter can be in a segment allocated by the Nucleus, but this is not a requirement.

NOTE

A call to A$READ will not be successful unless the mode of the open connection permits reading (see A$OPEN).

EXCEPTION CODES

A$READ can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection parameter is a buffered connection produced by the Extended I/O System.
E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter 2. The connection is being deleted.

EXCEPTION CODES (continued)

E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).
E$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	One or more of the following system calls was not included when the system was configured: A$READ GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	This connection is not open for read or update.
E$FLUSHING	The connection was closed before the READ request was completed.
E$IO	An I/O error occurred during the operation.
E$SPACE	The A$READ request attempted to read past the end of the physical device; this applies only to physical files.

EXCEPTION CODES (continued)

E$SUPPORT	Your system is configured incorrectly. The entry point associated with A$READ is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

A$RENAME$FILE

A$RENAME$FILE changes the path name of a named file.

```
CALL RQ$A$RENAME$FILE(connection, user, prefix, subpath, resp$mbox,
                      except$ptr);
```

INPUT PARAMETERS

connection	a WORD containing a token for a connection to the file being renamed; this connection and all other connections to the file will remain in effect after the file is renamed.
user	a WORD containing a token for the user object to be inspected in access checking; a zero specifies the default user for the job.
prefix	a WORD containing a token for the connection to be used as the starting point in a path scan; a zero specifies the default prefix for the job.
subpath	a POINTER to a STRING containing the new subpath for the file; prefix and subpath must not lead to an already-existing file; the string pointed to by the subpath parameter cannot be a null string.

OUTPUT PARAMETERS

resp$mbox	a WORD containing a token for the mailbox that receives an I/O result segment (see Appendix C) indicating completion of the rename operation. A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$RENAME$FILE system call applies to named files only. It is called to change the path name of a file. For named data or directory files, A$RENAME$FILE can be used to recatalog files in different parent directories, as long as the new directory is on the same volume as the file's original parent directory.

There is one restriction governing the renaming of a directory. Any attempt to rename a directory as its own parent will cause the Basic I/O System to return an exception code. Also, be aware that renaming a directory changes the paths of any files contained in the directory.

NOTE

The caller must have delete access to the original file and must have add-entry access to the file's parent directory.

EXCEPTION CODES

A$RENAME$FILE can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection and the prefix in the call refer to different devices. You cannot simultaneously rename a file and move it to another device.
EDEVOFFLINE	The prefix parameter in this system call refers to a logical connection. Either: • The device is offline, or • The device has never been physically attached. (See Appendix E for a more detailed explanation.)

EXCEPTION CODES (continued)

E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The prefix parameter • The connection parameter • The response mailbox parameter • The user parameter. 2. The prefix connection is being deleted.
E$IFDR	This system call applies only to named files, but the connection parameters specifies some other type of file.
E$LIMIT	Processing this call caused one of these limits to be exceeded: • The maximum number of objects allowed for this job (specified when the job was created). • The number of I/O operations which can be outstanding at one time for the user object specified in the call (255 decimal).
E$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	You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following: • When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default prefix. • No default prefix is cataloged for this job. • When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.
ENOUSER	If the user parameter in this call is not zero, then the problem is that the parameter is not a user object.

EXCEPTION CODES

ENOUSER (continued)

If the user parameter is zero, it specifies a default user. But no default user can be found because:

- When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default user.
- No default user is cataloged for this job.
- When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.
- The object which is cataloged with the name R?USER is not a user object. The name R?USER should be treated as a reserved word.

ENOTCONFIGURED — One or more of the following system calls was not included when the system was configured:

RENAME$FILE
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)
DISABLE$DELETION (Nucleus)

E$PARAM — The path name contains invalid characters, or has a length of zero.

E$SUPPORT — The connection parameter specified is not valid in this system call because the connection was not created by this job.

E$TYPE — One of three conditions caused this exception:

- The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.)
- The connection parameter is not a connection object.
- The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	One of these conditions caused this exception code: • The file specified is on a device which the system is detaching. • The call is attempting to rename the ROOT directory. • The call is attempting to rename the directory to a new path containing itself. This is specifically forbidden; see DESCRIPTION.
E$FACCESS	Either: • The user object in the parameter list is not qualified for "delete" access to the file, or • The connection doesn't have "add entry" access to the parent directory.
E$FEXITS	The file name already exists.
E$FNEXIST	This indicates one of the following circumstances: • A file in the specified path does not exist. • A file in the specified path is marked for deletion.
E$FTYPE	The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last string, each string in a pathname must be a named directory.)
E$IO	An I/O error occurred during the operation.
E$LIMIT	To service this call, the Basic I/O System had to create some objects. The maximum number of objects which it can contain at one time was exceeded. (This maximum limit is specified when the Basic I/O System job is configured.)
E$MEM	The memory pool of the Basic I/O System does not currently have a block of memory large enough to allow this system call to run to completion.

EXCEPTION CODES (continued)

E$SPACE	There is no more space on this volume.
E$SUPPORT	As the Basic I/O System is configured, either: • The entry point associated with A$RENAME is not included in the "I/O System part" of the named file driver table. The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information. • The Basic I/O System does not allow allocation of space on volumes.

A$SEEK

A$SEEK moves the file pointer of an open connection.

```
CALL RQ$A$SEEK(connection, mode, hi$ptr$move, low$ptr$move, resp$mbox,
                except$ptr);
```

INPUT PARAMETERS

connection — a WORD containing a token for the open file connection whose file pointer is to be moved.

mode — a BYTE describing the movement of the file pointer; possible values are:

1. Move pointer back by "ptr$move" amount; if this action moves the pointer past the beginning of the file, the pointer is set to zero (first byte).
2. Set the pointer to the location specified by "ptr$move".
3. Move the file pointer forward by "ptr$move" amount.
4. Move the pointer to the end of the file, minus the "ptr$move" specified.

hiptrmove
lowptrmove — a pair of words giving the number of bytes involved in the seek; the interpretation of "ptr$move" depends on the mode setting, as just explained.

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives an I/O result segment (see Appendix C) when the seek is completed. A value of zero means that you do not want to receive an I/O result segment.

If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$SEEK system call applies to physical and named files only. This call moves the file pointer for an open connection, thus allowing file contents to be accessed randomly. The file pointer can be moved to any byte position in the file; the first byte is byte zero.

EXCEPTION CODES

A$SEEK can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection parameter is a buffered connection produced by the Extended I/O System.
E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter. 2. The connection is being deleted.
E$IFDR	This system call applies only to named and physical files, but the prefix and subpath parameters specify a stream file.
E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).

EXCEPTION CODES (continued)

E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
ENOTCONFIGURED	One or more of the following system calls was not included when the system was configured: A$SEEK GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$PARAM	The mode parameter value is out of the valid range (1 to 4).
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection is not open.
E$FLUSHING	The connection specified in the call was closed before the seek operation could be completed.
E$IO	An I/O error occurred during the operation.
E$PARAM	This call attempted to seek beyond the end of the physical device. This applies only to physical files.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with A$SEEK is not included in the "I/O System part" of the file driver table (named or physical file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

ASETEXTENSION$DATA

The ASETEXTENSION$DATA system call writes the extension data for a Basic I/O System file.

```
CALL RQ$A$SET$EXTENSION$DATA(connection, data$ptr, resp$mbox,
                             except$ptr);
```

This System Programmer call is included here for convenience. ASETEXTENSION$DATA is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

A$SPECIAL

A$SPECIAL enables tasks to perform a variety of special functions.

```
CALL RQ$A$SPECIAL(connection, spec$func, ioparm$ptr, resp$mbox,
                  except$ptr);
```

INPUT PARAMETERS

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

spec$func — an encoded WORD that, with the connection argument, specifies the function being requested; the functions are described under the heading DESCRIPTION and are summarized as follows:

file driver for connection	spec$func value	function
physical	0	format track
physical or named	2	notify
stream	0	query
stream	1	satisfy

ioparm$ptr — a POINTER to a parameter block; the contents of the parameter block depends upon the requirements of the special function being requested and are described fully under the heading DESCRIPTION.

OUTPUT PARAMETERS

resp$mbox — a WORD containing a token for the mailbox that receives the I/O result segment (see Appendix C) for the special operation. A value of zero means that you do not want to receive an I/O result segment.

If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.

except$ptr — a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$SPECIAL system call enables tasks to perform a variety of special functions.

Tasks define their requests by means of the spec$func and ioparam$ptr parameters. Spec$func is a code which, when combined with the file driver associated with the connection argument, specifies the function the Basic I/O System is to perform. When more information is needed to define a request, ioparam$ptr points to a parameter block containing the additional data. Descriptions of the available functions follow.

Formatting a Track. This function applies to physical files only. To format a track on a flexible diskette, set spec$func to 0, and set ioparam$ptr to point to a structure of the form

```
DECLARE format$track  STRUCTURE(
    track$number    WORD,
    interleave      WORD,
    track$offset    WORD);
```

where:

track$number	the number of the track to be formatted; acceptable values are 0 to 76; other values cause an E$SPACE exceptional condition.
interleave	the interleaving factor for the track (that is, the number of physical sectors to skip before locating the next logical sector); the supplied value, before being used, is evaluated mod 26 for 128-byte sectors or mod 8 for 512-byte sectors.
track$offset	the number of physical sectors to skip before locating logical sector one.

To format a track on a hard disk, set spec$func to 0 and set ioparam$ptr to point to a structure of the form:

```
DECLARE format$track  STRUCTURE(
   track$number     WORD,
   interleave       WORD,
   track$offset     WORD,
   fill$char        WORD);
```

where:

track$number	the number of the track to be formatted; acceptable values are 0 to 799; other values cause an E$SPACE exceptional condition.

DESCRIPTION (continued)

interleave	the interleaving factor for the track; the supplied value, before being used, is evaluated mod 36 for 128-byte sectors or mod 12 for 512-byte sectors.
track$offset	the number of physical sectors to skip before locating logical sector one.
fill$char	the byte value with which each sector is to be filled.

Requesting Notification that a Volume is Unavailable. This function applies to named and physical files only.

When a person opens a door to a flexible disk drive or presses the STOP button on a hard disk drive, the volume mounted on that drive becomes unavailable. A task can request notification of such an event by calling A$SPECIAL. For flexible disk drives attached to an iSBC 204 controller, notification occurs when the Basic I/O System first tries to perform an operation on the unavailable volume. For most other drives notification occurs immediately. The reason for this difference is that the iSBC 204 controller does not generate an interrupt when its drives cease to be ready. In contrast, most other controllers do.

To request notification, a task calls A$SPECIAL with a token for a device connection, with spec$func set to 2, and with ioparam$ptr pointing to a structure of the form:

```
DECLARE notify  STRUCTURE(
   mailbox    WORD,
   object     WORD);
```

where:

mailbox	contains a token for a mailbox.
object	contains a token for an object; when the Basic I/O System detects that the implied volume is unavailable, the object is sent to the mailbox.

After a task has made a request for notification, the Basic I/O System remembers the object and mailbox tokens until either the volume is detected as being unavailable or until the device is detached (see A$PHYSICAL$DETACH$DEVICE in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL). When the volume becomes unavailable, the object is sent to the mailbox. Note that this implies that some task should be dedicated to waiting at the mailbox.

DESCRIPTION (continued)

If the volume is detected as being unavailable, the Basic I/O System will not execute I/O requests to the device on which the volume was mounted. Such requests are returned with the status field of the I/O result segment set to E$IO and the unit$status field set to IO$OPRINT (value = 3). The latter code means that operator intervention is required.

To restore the availability of a volume, four steps are required:

1. Close the door of the diskette drive or restart the hard disk drive.

2. Call A$PHYSICAL$DETACH$DEVICE (see the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL). It may be necessary to do a "hard" detach of the device.

3. Call A$PHYSICAL$ATTACH$DEVICE (see the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL) and reattach the device.

4. Create a new file connection.

To cancel a request for notification, make a dummy request using the same connection with a 0 value in the mailbox parameter. In this case, the result is that there is no longer a request for notification.

Obtaining Information about Stream File Requests. Occasionally, a task using a stream file needs to know what is being requested by the other task using the same stream file. For example, the task doing a read operation on a stream file might need to know how many bytes are being sent by the task doing a write operation on the same file. Tasks can obtain this kind of information by calling A$SPECIAL, using the connection for the stream file, with spec$func set to 0 (query). The ioparam$ptr argument is ignored.

If a read or write request is queued at the file, the information requested is returned in the I/O result segment for the call to A$SPECIAL. The actual field contains the number of bytes being sent, the count field contains the number of bytes still remaining in the buffer, and the buff$ptr field points to the buffer.

If no read or write request is queued at the file, the calling task's request for information is queued at the file. If a second request for information is made before the first one is satisfied, the I/O result segments for both requests are returned with E$CONTEXT in the status field.

Artificially Satisfying a Stream File I/O Request

When a task tries to read or write to a stream file, the request is not satisfied until the other task makes a request that matches the first request. For example, if task A wants to read 512 bytes, but task B only wants to write 256 bytes, only 256 bytes are transferred. Task A continues to wait for the other 256 bytes, even though Task B may never write them.

By using A$SPECIAL, with a stream file connection and with spec$func set to 1 (ioparam$ptr is ignored), either task can force the data to transfer request to be satisfied, even though the reading task is requesting more bytes than the writing task is providing. After the transfer, the tasks can ascertain the number of bytes sent by checking the actual field in their respective I/O result segments.

A task trying to satisfy an I/O request in this way will receive an E$CONTEXT exceptional condition if no request is queued at the stream file or if a request for information is queued. In the latter case, the task that submitted the request for information also receives an E$CONTEXT condition.

EXCEPTION CODES

A$SPECIAL can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection parameter is for a buffered I/O connection, which is not valid in this call. (Buffered I/O connections are a function of the Extended I/O System.)
E$EXIST	Two conditions can cause this exception code to be returned:

EXCEPTION CODES

E$EXIST (continued)

1. At least one of the following parameters is not a token for a valid object:
   - The connection parameter
   - The response mailbox parameter
2. The connection is being deleted.

E$IFDR — The function requested (spec$func) is not valid for the type of file specified by the connection parameter.

E$LIMIT — The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).

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 — One or more of the following system calls was not included when the system was configured:

SPECIAL
GET$TYPE (Nucleus)
SEND$MESSAGE (Nucleus)
CREATE$SEGMENT (Nucleus)

E$PARAM — The special request applies to a stream file, but the code is not either "query" or "notify".

E$SUPPORT — The connection parameter specified is not valid in this system call because the connection was not created by this job.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK — NORMAL CODE. No exceptional conditions.

E$CONTEXT — One of the following exceptional conditions exists:

- The function code is 'notify' and the connection is not a device connection. This applies to named and physical files only.

EXCEPTION CODES (continued)

- The connection is not open. This applies to stream and physical files only.
- This is a "query" request, but another query is already queued This applies only to stream files.
- This is a "satisfy" request, but either a query request is queued, or no requests are queued. This applies only to stream files. (See Artifically Satisfying A Stream File I/O Request.)

E$FLUSHING	The connection to which this special function applies was closed before the function could be completed.
E$IDDR	The function being requested is not valid for the device specified by the connection parameter.
E$IFDR	The connection refers to a named file, but the function is not "notify".
E$IO	An I/O error occurred during the operation.
E$SPACE	The A$SPECIAL call attempted to format a physical file past the end of the device.
E$SUPPORT	Your system is configured incorrectly. The entry point associated with A$SPECIAL is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.

A$TRUNCATE

A$TRUNCATE truncates a named file at the current setting of the pointer, freeing all allocated space beyond the pointer.

```
CALL RQ$A$TRUNCATE(connection, resp$mbox, except$ptr);
```

INPUT PARAMETER

connection	a WORD containing a token for an open connection to the file being truncated.

OUTPUT PARAMETERS

resp$mbox	a WORD containing a token for the mailbox that receives the I/O result segment (see Appendix C) for the truncate operation. A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$TRUNCATE system call applies to named files only. This call truncates a file at the current setting of the file pointer, freeing all allocated space beyond the pointer. A$SEEK can be called to position the pointer before A$TRUNCATE is called. If the file pointer is at or beyond the end-of-file, no operation is performed.

Truncation is performed immediately, rather than waiting until connections to the file are deleted.

NOTE

The designated file connection must be open for writing and the connection must have update access to the file.

EXCEPTION CODES

A$TRUNCATE can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection parameter is a buffered I/O connection, which is invalid here. (Buffered I/O connections are a function of the Extended I/O System.
E$EXIST	Two conditions can cause this exception code to be returned: 1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter 2. The connection is being deleted.
E$IFDR	This system call applies only to named files, but the parameter list specified some other type of file.
E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).
E$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.

EXCEPTION CODES (continued)

ENOTCONFIGURED	One or more of the following system calls was not included when the system was configured: A$TRUNCATE GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The file specified is not open for write or update.
E$FACCESS	The connection in the parameter list is not qualified for "update" access to the file.
E$IO	An I/O error occurred during the operation.
E$SUPPORT	As the Basic I/O System is configured, either: • The entry point associated with A$TRUNCATE is not included in the "I/O System part" of the named file driver table. The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information, or • Truncating files is not allowed.

A$WRITE

A$WRITE writes data from the calling task's buffer to a connected file.

```
CALL RQ$A$WRITE(connection, buff$ptr, count, resp$mbox, except$ptr);
```

INPUT PARAMETERS

connection	a WORD containing a token for the open connection through which the write operation is to take place.
buff$ptr	a POINTER to the buffer (segment) that contains the data to be written.
count	a WORD giving the number of bytes to be written.

OUTPUT PARAMETERS

resp$mbox	a WORD containing a token for the mailbox that receives the I/O result segment (see Appendix C) for the write operation. A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should issue DELETE$SEGMENT to delete the segment. If all the other connections to a stream file are requesting write operations, an actual value of zero and a status value for E$FLUSHING are returned in the I/O result segment.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The A$WRITE call writes data from the caller's buffer to a connected file. The data is written starting at the current file pointer. After the write operation, the file pointer is positioned just after the last byte written. Some efficiency may be gained by starting writes on device block boundaries and writing an integral number of device blocks.

DESCRIPTION (continued)

Be aware that it is possible to use the A$SEEK system call to position the file pointer beyond the end of the file and comence writing. If a task does this, the Basic I/O System will extended the file to accommodate the writing operation. However, the positions in the file located between the old end of file and the beginning of the writing operation will contain arbitrary information.

NOTES

The buffer supplying the data to be written should not be modified until the write request has been acknowledged at the response mailbox.

The designated file connection must be open for writing, and the connection must have append or update access to the file.

EXCEPTION CODES

A$WRITE can return exception codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential code. A code returned as a result of asynchronous processing is a concurrent exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual (ASYNCHRONOUS SYSTEM CALLS).

The following list is divided into two parts -- one for sequential codes, and one for concurrent codes.

Sequential Exception Codes

The Basic I/O System can return the following exception codes to the WORD specified by the except$ptr parameter of this system call.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection specified is a buffered I/O connection, which is not valid for the A$WRITE call. (Buffered I/O connections are a function of the Extended I/O system.)
E$EXIST	Two conditions can cause this exception code to be returned:

EXCEPTION CODES (continued)

	1. At least one of the following parameters is not a token for a valid object: • The connection parameter • The response mailbox parameter 2. The connection is being deleted.
E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).
E$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	One or more of the following system calls was not included when the system was configured: A$WRITE GET$TYPE (Nucleus) SEND$MESSAGE (Nucleus) CREATE$SEGMENT (Nucleus)
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One of two conditions caused this exception: • The connection parameter is not a valid object type. It must be a connection object. • The response mailbox parameter in the call is not a token for a mailbox.

Concurrent Exception Codes

The Basic I/O System can return the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	The connection is not open for update or write.
E$FACCESS	The connection in the parameter list is not qualified for "update" or "append" access to the file.

EXCEPTION CODES (continued)

E$FLUSHING	Either: • The connection was closed before the write could be performed, or • The file specified by the connection parameter is a stream file, and all other connections are also requesting to write the file. (See the description of resp$mbox.)
E$IO	An I/O error occurred during the operation.
E$SPACE	Either: • The volume has no more space, or • The operation attempted to write beyond the end of the device. This applies to physical files only.
E$SUPPORT	As the Basic I/O System is configured, either: • The entry point associated with A$WRITE is not included in the "I/O System part" of the file driver table (named, physical, or stream file). The corresponding entry point in the "Request part" is included. Refer to the iRMX 86 CONFIGURATION GUIDE for further information. • The write is attempting to extend the file, but allocation of file space is not allowed.

CREATE$USER

The CREATE$USER system call creates a user object.

```
user = RQ$CREATE$USER(ids$ptr, except$ptr);
```

This System Programmer call is included here for convenience. CREATE$USER is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

DELETE$USER

The DELETE$USER (Basic I/O) system call deletes a user object.

```
CALL RQ$DELETE$USER(user, except$ptr);
```

This System Programmer call is included here for convenience. DELETE$USER is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

GET$DEFAULT$PREFIX

GET$DEFAULT$PREFIX returns the default prefix of a job.

```
connection = GET$DEFAULT$PREFIX(job, except$ptr);
```

INPUT PARAMETER

job	a WORD containing a token for the job whose default prefix is sought; a zero specifies the calling job.

OUTPUT

except$ptr	a POINTER to a WORD where the sequential exception code will be returned.
connection	a WORD receiving a token for the connection object which is the default prefix for the designated job.

DESCRIPTION

The GET$DEFAULT$PREFIX system call allows the caller to determine the default prefix for the specified job.

EXCEPTION CODES

E$OK	NORMAL CODE. No exceptional conditions.
ENOPREFIX	You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following: • When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default prefix. • No default prefix is cataloged for this job. • When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.

EXCEPTION CODES

ENOPREFIX (continued)

- The prefix which is cataloged is not the correct type. The default prefix must be a connection object or logical device object. (Logical device objects are created by the Extended I/O System.)

ENOTCONFIGURED The system call GET$DEFAULT$PREFIX was not included when the Basic I/O System was configured.

GET$DEFAULT$USER

GET$DEFAULT$USER returns the default user object of a job.

```
user$id = GET$DEFAULT$USER(job, except$ptr);
```

INPUT PARAMETER

job	a WORD containing a token for the job whose default user object is sought; a zero specifies the calling job.

OUTPUT

except$ptr	a POINTER to a WORD where the sequential exception code will be returned.
user$id	a WORD containing a token for the user object which is the default user for the designated job.

DESCRIPTION

The GET$DEFAULT$USER system call allows the calling task to determine the default user object associated with the designated job.

EXCEPTION CODES

E$OK	NORMAL CODE. No exceptional conditions.
ENOUSER	No default user can be found because: • When this job was created, a size of zero was specified for its object directory. So the job cannot catalog a default user. • No default user is cataloged for this job. • When the system was configured, the Nucleus system call LOOKUP$OBJECT was not included.

EXCEPTION CODES

ENOUSER (continued)

- The object which is cataloged with the name R?USER is not a user object. The name R?USER should be treated as a reserved word.

ENOTCONFIGURED The system call GET$DEFAULT$USER was not included when the Basic I/O System was configured.

GET$TIME

GET$TIME system call returns the date/time value from its doubleword counter.

```
date$time = GET$TIME(except$ptr);
```

OUTPUT

date$time	a POINTER containing a date/time value expressed as the number of seconds since a fixed, user-determined point in time; the offset portion of the pointer contains the low part of the value and the base portion contains the high part of the value.
except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The GET$TIME system call returns the date/time value for the Basic I/O System. The Basic I/O System maintains the date/time value in two words containing the number of seconds since some fixed point in time. Any time in the past can be used as the "beginning of time". See your system programmer for the reference point used in your system.

EXCEPTION CODES

E$OK	NORMAL CODE. No exceptional conditions.
ENOTCONFIGURED	The system call GET$TIME was not included when the Basic I/O Sytem was configured.

INSPECT$USER

The INSPECT$USER (Basic I/O) System call returns a list of the ID's contained in a user object.

```
CALL RQ$INSPECT$USER(user, ids$ptr, except$ptr);
```

This System Programmer call is included here for convenience. INSPECT$USER is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

SET$DEFALULT$PREFIX

SET$DEFAULT$PREFIX sets the default prefix for an existing job.

```
CALL RQ$SET$DEFAULT$PREFIX(job, prefix, except$ptr);
```

INPUT PARAMETERS

job	a WORD containing a token for the job whose default prefix is to be set; a zero specifies the current job.
prefix	a WORD containing a token for the connection that is to become the default prefix.

OUTPUT PARAMETERS

except$ptr	a POINTER to a WORD where the condition code will be returned.

DESCRIPTION

The SET$DEFAULT$PREFIX system call sets the default prefix for an existing job. It does this by cataloging the connection (supplied as the prefix parameter) in the object directory of the job (supplied as the job parameter). The Basic I/O System catalogs the prefix under the name $.

EXCEPTION CODES

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	Either: • When this job was created, a size of zero was specified for the object directory. So a default prefix cannot be cataloged • As the system was configured, the Nucleus system call UNCATALOG$OBJECT was not included. So objects cannot be removed from the directory.

EXCEPTION CODES (continued)

E$EXIST	At least one of the following parameters is not a token for a valid object: • The prefix parameter • The job parameter
E$LIMIT	The prefix parameter cannot be cataloged because the job object directory is full.
ENOTCONFIGURED	One or more of the following system calls was not included when the system was configured: SET$DEFAULT$PREFIX CATALOG$OBJECT GET$TYPE (Nucleus)
E$TYPE	One of two conditions caused this exception: • The prefix parameter is not a valid object type. It must be either a connection object, or a logical device object (Logical devices are described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.) • The job parameter is not a token for a job.

SET$DEFAULT$USER

SET$DEFAULT$USER sets the default user object for a job.

```
CALL RQ$SET$DEFAULT$USER(job, user, except$ptr);
```

INPUT PARAMETERS

job	a WORD containing a token for the job whose default user object is to be set; a zero designates the calling task's job.
user	a WORD containing a token for the user object that is to become the default user.

OUTPUT PARAMETERS

except$ptr	a POINTER to a WORD where the sequential exception code will be returned.

DESCRIPTION

The SET$DEFAULT$USER system call sets the default user for an existing job.

EXCEPTION CODES

E$OK	NORMAL CODE. No exceptional conditions.
E$CONTEXT	Either: • When this job was created, a size of zero was specified for the object directory. So a default prefix cannot be cataloged, or • As the system was configured, the Nucleus system call UNCATALOG$OBJECT was not included. So objects cannot be removed from the directory.

EXCEPTION CODES (continued)

E$EXIST	At least one of the following parameters is not a token for a valid object: • The user parameter • The job parameter
E$LIMIT	The user object cannot be cataloged because the job object directory is full.
ENOTCONFIGURED	One or more of the following system calls was not included when the system was configured: SET$DEFAULT$USER CATALOG$OBJECT (Nucleus) GET$TYPE (Nucleus)
E$TYPE	The job or user argument refers to an object of the wrong type.

SET$TIME

The SET$TIME system call sets the date and time for the I/O System.

```
CALL RQ$SET$TIME(time$high, time$low,  except$ptr);
```

This System Programmer call is included here for convenience. SET$TIME is described completely in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL, Chapter 7.

APPENDIX A. iRMX™ 86 DATA TYPES

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

BYTE - An unsigned, one byte, binary number.

WORD - An unsigned, two byte, binary number.

INTEGER - A signed, two byte, binary number that is stored in two's complement form.

OFFSET - A word whose value represents the distance from the base of a segment.

TOKEN - A word whose value identifies an object.

POINTER - Two words containing the base of a segment and an offset, in the order: offset followed by base.

STRING - A sequence of consecutive bytes. The first byte contains the number of bytes that follow it in the string.

APPENDIX B. iRMX™ 86 TYPE CODES

Each iRMX 86 object type is known within the iRMX 86 System by means of a numeric code. For each code, there is a mnemonic name that can be substituted for the code. The following lists the types with their codes and associated mnemonics.

OBJECT TYPE	INTERNAL MNEMONIC	NUMERIC CODE
Job	T$JOB	001H
Task	T$TASK	002H
Mailbox	T$MAILBOX	003H
Semaphore	T$SEMAPHORE	004H
Segment	T$SEGMENT	006H
User	TNUMUSER	100H
Connection	T$CONNECTION	101H

APPENDIX C. I/O RESULT SEGMENT

Certain asynchronous I/O system calls return a data structure called an I/O result segment to the mailbox specified by the "resp$mbox" parameter. The following system calls can return such a segment:

A$ATTACH$FILE	A$CHANGE$ACCESS
A$CLOSE	A$CREATE$DIRECTORY
A$CREATE$FILE	A$DELETE$CONNECTION
A$DELETE$FILE	A$OPEN
A$READ	A$RENAME$FILE
A$SEEK	A$SPECIAL
A$TRUNCATE	A$WRITE

Three of these system calls (A$ATTACH$FILE, A$CREATE$DIRECTORY, and A$CREATE$FILE) can return either a connection or an I/O result segment to the mailbox. Your application task can determine which type of object has been returned by making a GET$TYPE system call before trying to examine the object.

Before waiting at the response mailbox to receive the I/O result segment, your application task should examine the condition code returned in the word pointed to by the "except$ptr" parameter. If this code is "E$OK", the task can wait at the mailbox. However, if the code is not "E$OK", an exceptional condition exists and nothing is sent to the mailbox.

Immediately after receiving the I/O result segment, the task should examine the status field. This field contains an "E$OK" if the system call was completed sucessfully or an exceptional-condition code if an error occurred. The result segment also contains the actual number of bytes read or written, if appropriate.

STRUCTURE OF I/O RESULT SEGMENT

The I/O result segment is structured as follows:

```
DECLARE   iors        STRUCTURE(
          status        WORD,
          unit$status   WORD,
          actual        WORD);
```

where:

status — the condition code indicating the outcome of the call; Appendix D lists these asynchronous condition codes.

unit$status — contains, in the low-order four bits, device-dependent error code information that is meaningful only if status = E$IO; the codes, their meanings, and their associated mnemonics are as follows:

code	mnemonic	meaning
0	IO$UNCLASS	Unclassifed error
1	IO$SOFT	Soft error; the I/O system has retried the operation and failed; another retry is not possible
2	IO$HARD	Hard error; a retry is not possible
3	IO$OPRINT	Operator intervention is required
4	IO$WRPROT	Write-protected volume

actual — the actual number of bytes transferred

The I/O result segment contains other fields which are of interest only to the designer of a device driver. These fields are not described in this manual. For further information about the remaining fields of the I/O result segment, refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM.

UNIT STATUS FOR SPECIFIC DEVICES

You may need to know the information contained in the "unit$status" field for the following devices.

iSBC 204 AND iSBC 206 CONTROLLERS

The iSBC 204 and 206 drivers place a controller-generated result byte in the high eight bits of this word. For information about this byte, refer to the hardware reference manual for the iSBC 204 or 206 controller.

iSBC 215 CONTROLLER

Under certain circumstances, the iSBC 215 Winchester disk controller places information in the high twelve bits of this word. If the low four bits indicate IO$SOFT, the controller sets the high twelve bits as follows:

Bit	Interpretation
15 (leftmost)	1=seek error
14	1=cylinder address miscompare
13	1=drive fault
12	1=ID field ECC error
11	1=data field ECC error
10-8	unused
7	1=sector not found
6-4	unused

On the other hand, if the low four bits indicate IO$HARD, the iSBC 215 controller sets the high twelve bits as follows:

Bit	Interpretation
15	1=invalid address
14	1=sector not found
13	1=invalid command
12	1=no index
11	1=diagnostic fault
10	1=illegal sector size
9	1=end of media
8	1=illegal format type
7	1=seek in progress
6	1=ROM error
5	1=RAM error
4	unused

If you need more detailed information regarding the meanings of these errors, refer to the iSBC 215 WINCHESTER DISK CONTROLLER HARDWARE REFERENCE MANUAL.

iSBC 208 CONTROLLER

If the error is IO$SOFT (low four bits =1H), the next hex digit position can be 0,1, or 2. That is, the value in the low byte of unit$status will be 01H, 11H, or 21H. The upper byte of the unit$status word will indicate the exact meaning of the error condition. The meanings are listed here.

low byte	high byte bit	meaning
01H	8,9	unit select
	10	head select
	11	not ready
	12	equipment check
	13	seek end
	14,15	
		00 normal termination
		01 abnormal termination
		10 invalid command
		11 ready state changed
11H	8	missing address mark
	9	not writeable
	10	no data, sector not found
	12	over-run, DMA late
	13	CRC error in ID field
	15	end of cylinder
21H	8	missing data address mark
	9	wrong cylinder in ID field
	12	wrong cylinder in ID field
	13	CRC error in data field
	14	deleted data mark

If you need more detailed information regarding the meanings of these errors, refer to the iSBC 208 FLEXIBLE DISK DRIVE CONTROLLER HARDWARE REFERENCE MANUAL.

APPENDIX D. EXCEPTION CODES

This Appendix lists two types of exception codes. Those detected synchronously with system call invocation (sequential codes) and those detected during the asynchronous portion of system call processing (concurrent codes). The sequential codes are returned to the location addressed by the "excep$ptr" field of the system call. The concurrent codes are returned in an I/O result segment (see Appendix C). This appendix lists all codes with their decimal and hexadecimal equivalents.

SYNCHRONOUS (ENVIRONMENTAL) EXCEPTION CODES

CODE	DECIMAL	HEXADECIMAL
E$OK	0	0H
E$TIME	1	1H
E$MEM	2	2H
E$LIMIT	4	4H
E$CONTEXT	5	5H
E$EXIST	6	6H
E$STATE	7	7H
ENOTCONFIGURED	8	8H
E$FEXIST	32	20H
E$FNEXIST	33	21H
E$SUPPORT	35	23H
E$FACCESS	38	26H
E$FTYPE	39	27H
E$SPACE	41	29H
EDEVOFFLINE	45	2DH

SEQUENTIAL (PROGRAMMER ERROR) EXCEPTION CODES

CODE	DECIMAL	HEXADECIMAL
E$ZERO$DIVIDE	32768	8000H
E$OVERFLOW	32769	8001H
E$TYPE	32770	8002H
E$PARAM	32772	8004H
EBADCALL	32773	8005H
E$IFDR	32800	8020H
E$NOUSER	32801	8021H
E$NOPREFIX	32802	8022H

CONCURRENT EXCEPTION CODES

CODE	DECIMAL	HEXADECIMAL
E$MEM	2	2H
E$LIMIT	4	4H
E$CONTEXT	5	5H
E$FEXIST	32	20H
E$FNEXIST	33	21H
E$DEVFD	34	22H
E$SUPPORT	35	23H
E$EMPTY$ENTRY	36	24H
EDIREND	37	25H
E$FACCESS	38	26H
E$FTYPE	39	27H
E$SHARE	40	28H
E$SPACE	41	29H
E$IDDR	42	2AH
E$IO	43	2BH
E$FLUSHING	44	2CH

APPENDIX E. LOGICAL DEVICES AND THE BASIC I/O SYSTEM

You can assign a logical name to any device with the system programmer call LOGICAL$ATTACH$DEVICE. This creates a Logical Device object, (TLOGDEV) and catalogs the object in the root object directory.

Typically, you will use these Logical Device objects with Extended I/O System calls. However, Basic I/O System calls also permit the prefix parameter to be a Logical Device object. When you use a Logical Device object as the prefix parameter in Basic I/O System calls, you might receive the exception code EDEVOFFLINE. If you receive this exception code and the device is online, the device was never physically attached.

Before you can use a logically named device, the device must be made known to the system (attached), with the Basic I/O System call A$PHYSICAL$ATTACH$DEVICE. But when LOGICAL$ATTACH$DEVICE is invoked, the system does not immediately issue a call to A$PHYSICAL$ATTACH$DEVICE. Instead, physical attachment occurs transparently during processing of any Extended I/O System call which references the Logical Device object.

You might create a logical device connection, but not invoke any Extended I/O System call to perform the physical attach operation. If so, the Basic I/O System will return EDEVOFFLINE. You can correct this situation by invoking at least one Extended I/O System call which uses the logical device.

For further information, refer to the descriptions of LOGICAL$ATTACH$DEVICE and A$PHYSICAL$ATTACH$DEVICE in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

INDEX

Underscored entries are primary references.

REQUEST FOR READER'S COMMENTS

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

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

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

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

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

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

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

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

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

O.M.S. Technical Publications

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

Printed in U.S.A.