intel®

iRMX™ 86 BASIC I/O SYSTEM REFERENCE MANUAL

iRMX™ 86 OPERATING SYSTEM

Order Number: 9803123-05

iRMX™86 BASIC I/O SYSTEM REFERENCE MANUAL

Order Number: 9803123-05

REV.	REVISION HISTORY	PRINT DATE
-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 Operating System. Change bars mark technical changes.	10/81
-05	Manual reorganized. System programmer information and system calls added. Configuration information added. Terminal Support Code information added. Terminal drive information added.	8/82

BXP	Intel	iSBC	Multibus
CREDIT	Int$_e$l	iSBX	Multichannel
i	Intelevision	iSXM	Multimodule
ICE	Intellec	Library Manager	Plug-A-Bubble
iCS	Intellink	MCS	PROMPT
im	iOSP	Megachassis	RMX/80
iMMX	iPDS	Micormainframe	System 2000
Insite	iRMX	Micromap	UPI

Printed in U.S.A./OM-014/5K/0383/S

PREFACE

This manual documents the Basic I/O System, one of the subsystems available with the iRMX 86 Operating System. Although the manual contains some introductory and overview material, it is intended primarily as a quick reference to system calls, providing detailed descriptions of those calls.

READER LEVEL

This manual is intended for 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 A$CREATE$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 PL/M-86 calling sequences.

You can also invoke the system calls 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.

- Introduction to the iRMX™ 86 Operating System, Order Number: 9803124
- iRMX™ 86 Operator's Manual, Order Number: 144523
- iRMX™ 86 Nucleus Reference Manual, Order Number: 9803122
- iRMX™ 86 Debugger Reference Manual, Order Number: 143323
- iRMX™ 86 Terminal Handler Reference Manual, Order Number: 143324
- iRMX™ 86 Programming Techniques Manual, Order Number: 142982
- Guide to Writing Device Drivers for the iRMX™ 86 and iRMX™ 88 I/O Systems, Order Number: 142926
- iRMX™ 86 Extended I/O System Reference Manual, Order Number: 143308
- iRMX™ 86 Configuration Guide, Order Number: 9803126
- PL/M-86 User's Guide, Order Number: 121636
- ASM86 Language Reference Manual for 8080/8085-Based Development Systems, Order Number: 121703
- ASM86 Macro Assembler Operating Instructions for 8086-Based Development Systems, Order Number: 121628

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

***

CHAPTER 1. ORGANIZATION

This manual is divided into nine chapters. Some of the chapters contain introductory or overview material that 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 that 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 through Chapter 6	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 7	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 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 9	This chapter lists the configuration options that pertain to the Basic I/O system. When configuring your software with the Interactive Configuration Utility, you define your system's requirements by specifying which of these options you want. These specifications are defined in the iRMX 86 CONFIGURATION GUIDE.

***

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 purposes of this chapter are to explain briefly 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 7.

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 several devices, including the following:

- iSBC 204 Single Density Flexible Disk Controller
- iSBC 206 Hard Disk Controller
- iSBC 208 Flexible Disk Controller
- iSBC 215 Winchester Hard Disk Controller
- iSBX 218 Multimodule Flexible Disk Controller
- iSBC 220 SMD Disk Controller
- iSBC 254 Bubble Memory Controller
- iSBX 270 Video Display Terminal Controller
- iSBC 534 Four-Channel Communications Expansion Board

- USART
- Byte Bucket
- Terminal or Teletypewriter

For a complete list of the provided drivers, see the iRMX 86 CONFIGURATION GUIDE.

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 and iRMX 88 I/O SYSTEMS.

THREE DISTINCT KINDS OF FILES

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 the indexed sequential access method (ISAM).

For more information regarding named files, refer to Chapter 4.

PHYSICAL FILES

Physical files differ from named files in that each physical file occupies an entire device. In fact, from the standpoint of the Basic I/O System, a physical file is a device. Yet with the Basic I/O System, an application can deal with a physical file as if it were a string of bytes.

Physical files provide several important advantages:

- An application can have direct control over a device.
- This direct control provides complete flexibility. For example, an application can interpret volumes created by other systems.
- An application can conserve memory and still be able to communicate with devices that do not need the power of named files. Examples of such devices include line printers, display tubes, plotters, and robots.

The disadvantages of physical files, as compared to named files, are that hierarchical file structures and access control are not available.

For more information about physical files, see Chapter 5.

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 information about stream files, see Chapter 6.

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 4, 5, and 6.

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 4.

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 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 4, 5, and 6.

CONTROL OVER FRAGMENTATION OF FILES

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. Similarly, the data on each mass storage volume, such as a flexible diskette, is divided into granules, whose size is called the volume granularity. In addition, the data on each mass storage device is divided into granules (called sectors in the case of disk media), whose size is the device granularity. The relationship between these three kinds of granularity are the following:

- The volume granularity is a multiple of the device granularity.
- The file granularity is a multiple of the volume granularity.

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.

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

***

CHAPTER 3. FUNDAMENTAL CONCEPTS

Before you use the Basic I/O System, you must understand several fundamental concepts. Some of those concepts were presented in Chapter 2. The remaining concepts are:

- System Programmers
- Device Controllers and Device Units
- 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 involves manipulating system resources and characteristics. These two roles are called application programming and system programming.

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

In Chapter 8 of this manual you will find several system call descriptions that begin with caution notices. These system calls, if misused, can have serious consequences for an application system. Therefore, you should consider these system calls to be reserved for the exclusive use of system programmers.

DEVICE CONTROLLERS AND DEVICE UNITS

You are undoubtedly familiar with the notion of a device. Devices include such things as flexible diskette drives, line printers, terminals, card readers, and the like. A device is a hardware entity that tasks can use to read information, to write information, or to do both.

In the iRMX 86 environment, it is convenient to make a distinction between devices and the hardware interfaces that communicate directly with an iRMX 86 application system. A hardware entity that talks directly with iRMX 86 software is a device controller. The kinds of things called devices in the previous paragraph are device units. Typically, a device controller interfaces between iRMX 86 application software and several device units. For example, an iSBC 220 SMD Disk Controller board acts as an interface between application software and from one to four Storage Module Devices (device units.)

VOLUMES

A volume is the medium used to store the information on a device unit. For example, if the device unit is a flexible disk drive, the volume is a diskette; if the device unit is a magnetic tape drive, the volume is the reel of tape; and if the device unit is a multiplatter hard disk drive, the volume is the disk pack.

FILES

Some operating systems treat a file as a device, while others treat a file as 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, and each has characteristics that make it unique. These characteristics are described in general in Chapter 2 and in detail in Chapters 4, 5, and 6.

Regardless of the kind of file, the Basic I/O System provides information to applications as a string of bytes, rather than as a collection of records.

CONNECTIONS FOR COMMUNICATION BETWEEN TASKS AND DEVICE UNITS

In complex environments such as those supported by the iRMX 86 Operating System, several layers of software and hardware must be bound together before communication between application tasks and device units can commence. Figure 3-1 shows these layers.

INTERLAYER BONDS PRECEDING INITIALIZATION

The bond between a device controller and the device units that it controls is a physical bond, usually in the form of wires or cables.

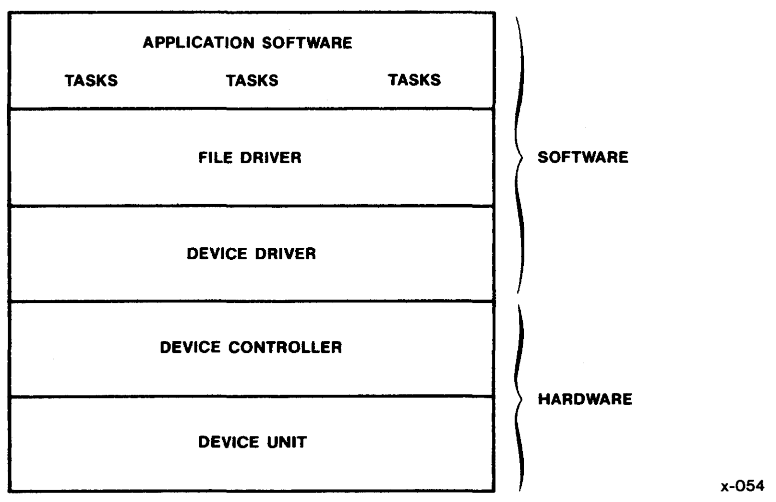


Figure 3-1. Layers of Interfacing Between Tasks and a Device

A device driver is bound to device controllers by data residing in a data structure known as a Device Unit Information Block (DUIB). (DUIBs are described fully in the Guide to Writing Device Drivers for the iRMX 86 and iRMX 88 I/O Systems.) You supply the data for the DUIBs when configuring the Operating System. (See the iRMX 86 CONFIGURATION GUIDE.)

When your application starts up, there is a gap between the application software and the file drivers, and another gap between the file drivers and the device drivers. Figure 3-2 illustrates this situation. The new element, shown in the figure as the configuration interface, is the "glue" that provides the final bonds.

POST-INITIALIZATION BOND -- THE CONFIGURATION INTERFACE

The configuration interface provides two kinds of system calls. Before a task can use a file, both of these kinds of calls must be invoked, and each produces a connection. These connections are called device connections and file connections, and several of them are shown in Figure 3-3 as conduits and wires through the conduits, respectively.

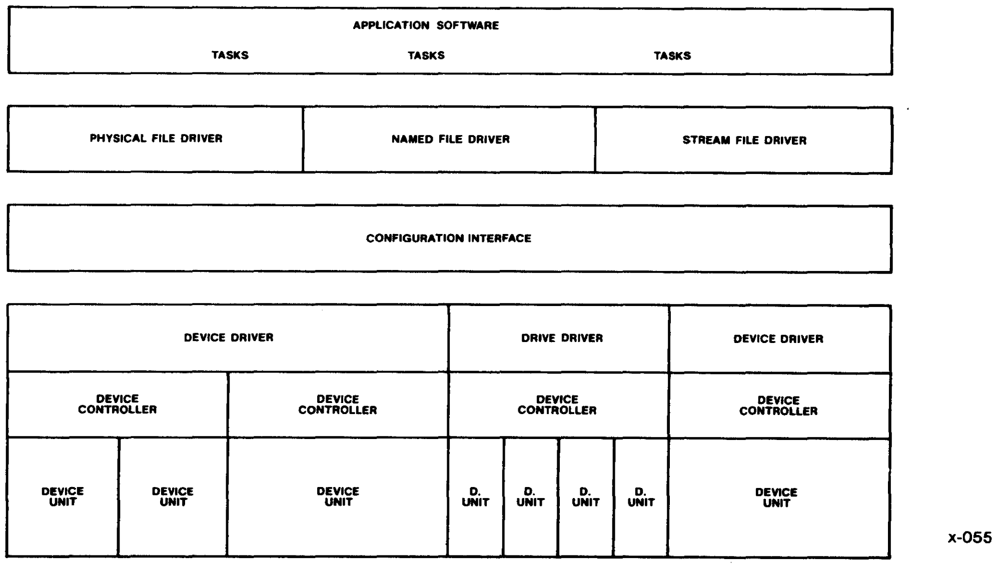


Figure 3-2. Schematic of Software at Initialization Time

Device Connections

Tasks employ the configuration interface first by calling the A$PHYSICAL$ATTACH$DEVICE system call, which returns a token for an iRMX 86 object type called a device connection. This device connection is the application's only pathway to the device. Moreover, there can be only one device connection between a device unit and all of the application tasks that need to use the device.

Because the device connection is so centrally important to the application, only tasks written by a system programmer should call A$PHYSICAL$ATTACH$DEVICE. Such a task could make the device connection available to application tasks selectively by sending it to certain mailboxes or by cataloging it in certain object directories. Or, to ensure that all required device connections will be available to all of the application tasks that need them, the system programmer could provide an initialization task that creates all of those device connections and catalogs them in the root object directory.

If and when the device is no longer needed by the application, an appropriate task can call A$PHYSICAL$DETACH$DEVICE to delete the device connection.

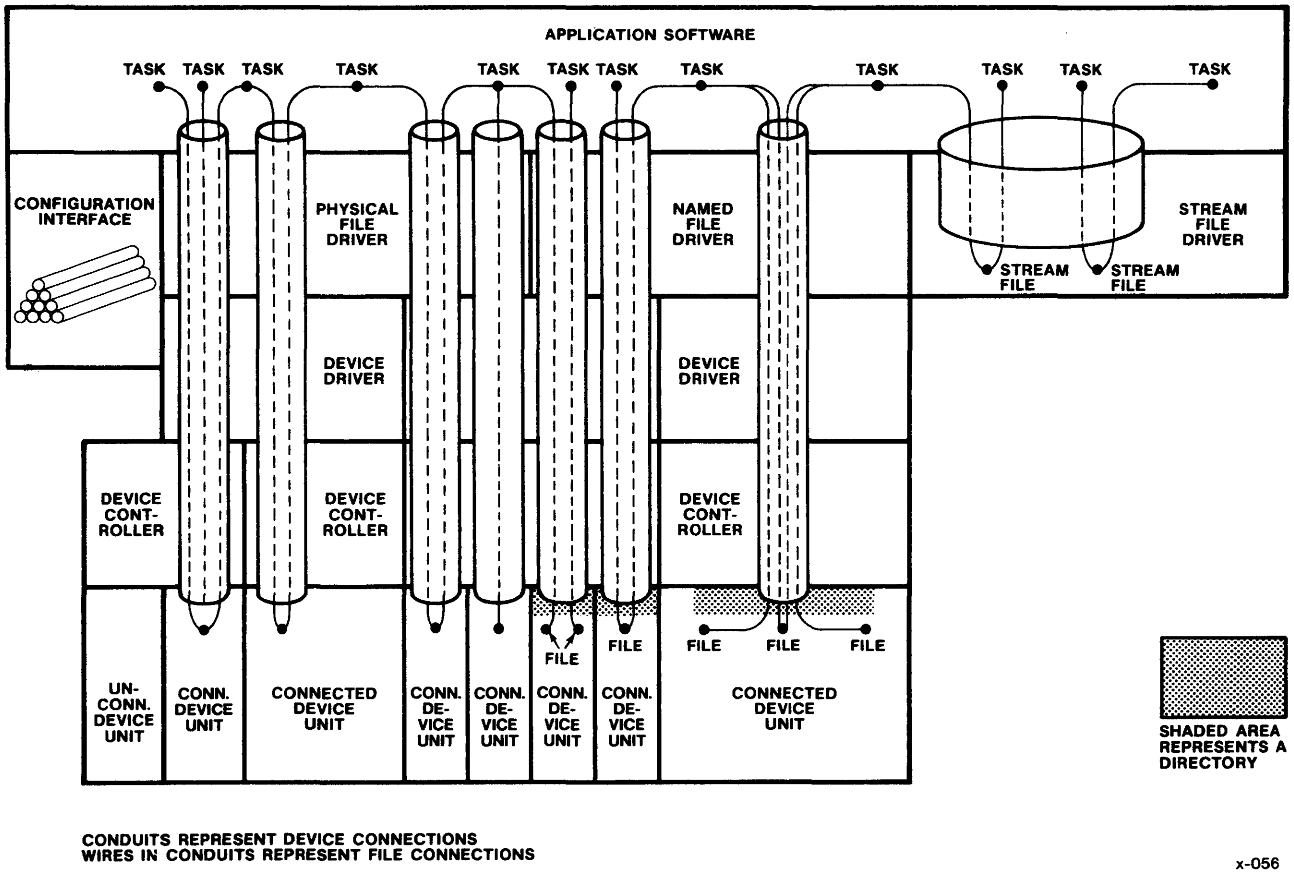


Figure 3-3. A System with Device and File Connections

File Connections

When an application task is ready to use a device unit, it must use the device connection for that device unit to obtain a <u>file connection</u> object, which is a connection to a particular file on that device unit. How the task does this depends on whether the file already exists. If the file already exists, the task usually calls A$ATTACH$FILE, although it can also call A$CREATE$FILE. If the file does not yet exist, the task must call A$CREATE$FILE.

NOTE

Even though a task can call A$CREATE$FILE to obtain a file connection for a file that already exists, it is not a good idea for a task to use A$CREATE$FILE unless the task is certain that the file does not yet exist. There are two reasons for this.

First, if a named file exists, then calling A$CREATE$FILE to obtain a connection to the file might cause the file to be truncated. This could cause problems for tasks having other connections to that file, because the file pointers (discussed later in this section) for those other connections are not affected, even though the end-of-file marker might be moved closer to the beginning of the file.

Second, if a file exists as either a physical or stream file, then it does not matter whether new connections to the file are obtained by a call to A$CREATE$FILE or A$ATTACH$FILE. However, it is possible that the code that does this will someday be used to create a connection to a named file, and as you can see, this can cause problems.

Unlike device connections, there can be multiple file connections to a file. This allows different tasks, if necessary, to have different kinds of access to the same file at the same time, as the next paragraph shows.

After receiving a file connection, a task calls A$OPEN to open the connection. In the call to A$OPEN, the task specifies how it intends to use the file connection and how it is willing to share the file with other tasks using other connections, by passing the following as parameters:

- An open-mode indicator

  The open-mode indicator tells the Basic I/O System how your application is going to use the connection. This parameter can specify that the connection is open for reading only, for writing only, or for both reading and writing.

- A share-mode indicator

  The share-mode indicator specifies how other connections can share the file with the connection being opened. This parameter can specify that there can be no other open connections to the file, that other connections to the file can be opened for reading only, that other connections to the file can be opened for writing only, or that other connections to the file can be opened for both reading and writing.

For each open file connection to a random-access device unit, the Basic I/O System maintains a file pointer. This is a number that tells the Basic I/O System the logical address of the byte where the next I/O operation on the file is to begin. The logical addresses of the bytes in a file begin with zero and increase sequentially through the entire file. Normally the pointer for a file connection points at the next logical byte after the one most recently read or written. However, a task can use the file connection, if need be, to modify the file pointer by means of the A$SEEK system call.

Some Observations about Devices and Connections

Figure 3-3 is quite detailed and shows most of the situations that are possible for device units and file connections to them. In particular, you can observe the following:

- Device connections extend from the application software to the individual device units, and each passes through one and only one file driver.

- There is only one device connection to each connected device, and multiple file connections can share the same device connection.

- Different device units with the same controller can be connected via different file drivers.

- Tasks can share access to the same device unit through the physical file driver, and they can share access to the same files on the same device unit through the named file driver.

- There is only one device connection through the stream file driver, reflecting the fact that a single, logical device contains all stream files. There _can_ be additional stream files in the application if more are needed.

- The configuration interface, which is depicted as a pile of conduits, is off to one side.

- All but one of the device units are connected. The unconnected device unit is still separated from the application software by the configuration interface.

***

CHAPTER 4. 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
- Extra Data in a File's Descriptor

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 4-1, 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 4-1. 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 4-1) 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 flexibility in organizing your file structure.

To illustrate this flexibility, take a close look at Figure 4-1. This figure shows how named files can be useful in multi-user systems. Figure 4-1 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.

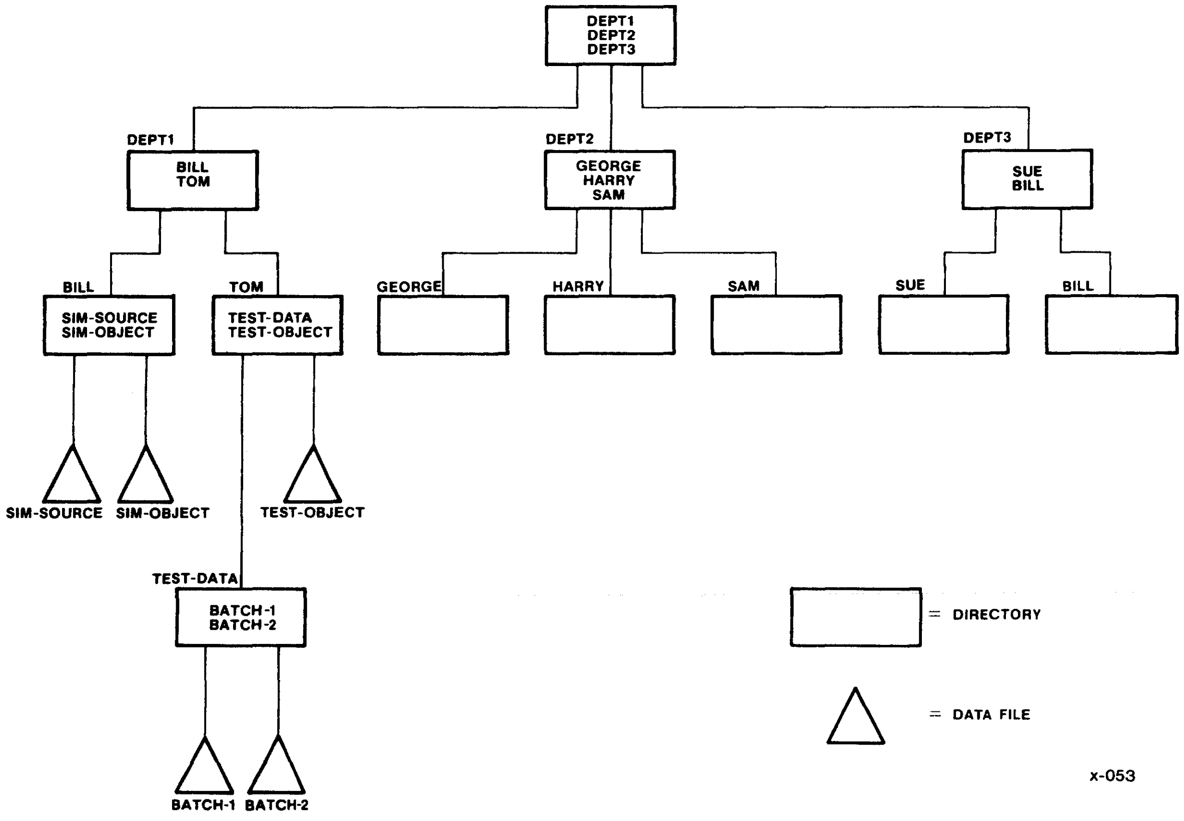


Figure 4-1. Example of a Named-File Tree

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 4-1, the file called SIM-SOURCE has the path DEPT1/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 4-1 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:

DEPT1/BILL/SIM-OBJECT

whereas the rightmost SIM-OBJECT would be identified by

DEPT3/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 <u>path name</u>. 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 DEPT2/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 4-1, suppose your application has a connection to Directory DEPT1/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 DEPT1/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 DEPT1/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 4-1, 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 4-1, Tom's application would probably use a connection to Directory DEPT1/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 programming 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.

CONTROLLING ACCESS TO FILES

In most environments where files are shared among multiple users, it is necessary to have a means of controlling which users have access to which files. And among users who have access to a given file, it is frequently necessary to grant different kinds of access to different users. This section describes the features of the Basic I/O System that can be used to ensure that only the intended type of access is granted to each user or class of user.

TYPES OF ACCESS TO FILES

Each of the two kinds of named files -- directory files and data files -- can be accessed in four different ways.

Every directory file can potentially be accessed in one or more of the following ways:

Delete	Delete the directory file with A$DELETE$FILE.
List	Obtain the contents of the directory file with A$READ or A$GET$DIRECTORY$ENTRY.
Add Entry	Add entries to the directory with A$CREATE$FILE, A$CREATE$DIRECTORY, or A$RENAME$FILE.
Change Entry	Change the access rights (explained shortly) of files listed in the directory with A$CHANGE$ACCESS.

Every data file can potentially be accessed in one or more of the following ways:

Delete	Delete the file with A$DELETE$FILE or rename the file with A$RENAME$FILE.
Read	Read the file with A$READ.
Append	Add information to the end of the file with A$WRITE.
Update	Change information in the file with A$WRITE or drop information with A$TRUNCATE.

A compact means of describing the kinds of access permitted a user of a file is an <u>access mask</u>. An access mask is a byte in which individual bits are used to represent the various kinds of access permitted or denied that user. When such a bit is set to 1, it signifies that the associated kind of access is permitted. When set to 0, the bit signifies that the associated kind of access is denied.

The association between the bits of the access mask and the kinds of access they control are as follows:

Bit	Directory Files	Data Files
0	Delete	Delete
1	List	Read
2	Add Entry	Append
3	Change Entry	Update

The remaining bits in the access mask have no significance.

ID NUMBERS

An ID number, or <u>ID</u> for short, is a 16-bit number that represents any individual or collection of individuals requiring a separate identity for the purpose of gaining access to files.

USER OBJECTS

The Basic I/O System uses a special type of object called a user object when determining access rights to files. A user object consists of one or more IDs.

The Basic I/O System has three system calls that manipulate user objects:

- CREATE$USER creates a user object and returns to the calling task a token for that user object.
- DELETE$USER deletes a user object.
- INSPECT$USER returns to the calling task the list of IDs in the user object specified in the call.

DEFAULT USER OBJECT FOR A JOB

In applications using the Basic I/O System, each job can have a default user object. Tasks in the job can specify this default user object in certain system calls simply by passing a zero value as a user object parameter.

SET$DEFAULT$USER can be used either to change an existing default user object or, in the case of jobs having no default user object, to establish one. GET$DEFAULT$USER can be used to ascertain the default user for a job.

FILE ACCESS LISTS

The access list for a file is a collection of up to three pairs of IDs and access masks. The IDs represent users or collections of users, and the access masks specify the kinds of access to the file that those users or collections of users are allowed.

For example, an access list for a data file might look like the following:

```
5B31  00001110
9F2C  00000010
```

where the ID numbers (left column) are in hexadecimal and the access masks (right column) are in binary. This means that the the ID number 5B31 has read, append, and update access rights, while the ID number 9F2C has the read access right.

The first entry in a file's access list is placed there automatically when the file is created. The ID portion of that entry is the first ID number in the user object specified in the call to A$CREATE$FILE and is known as the owner ID for the file. The access mask portion is supplied as a parameter in that same call.

Tasks can alter the access list of a file by means of the A$CHANGE$ACCESS system call. With A$CHANGE$ACCESS, ID-access pairs can be added or deleted, and the access masks for IDs already in the list can be changed.

ACCESS MASKS FOR FILE CONNECTIONS

Whenever a task calls A$CREATE$DIRECTORY, A$CREATE$FILE, or A$ATTACH$FILE, the Basic I/O System constructs an access mask and binds it to the file connection object returned by the call. This access mask is constant for the life of the connection, even if the access list for the file is subsequently altered. When the connection is used to manipulate the file, the access mask for the connection determines how the file can be accessed. For example, if the computed access rights for a connection to a data file do not include appending or updating, then that connection cannot be used in an invocation of A$WRITE.

Figure 4-2 illustrates the algorithm used to construct the access mask that is computed for a file connection when that connection is created by means of a call to A$CREATE$FILE or A$ATTACH$FILE. As the figure shows, the Basic I/O System compares the IDs in the specified user object with the IDs in the file's access list. The access masks corresponding to matching IDs are logically ORed, forming an aggregate mask.

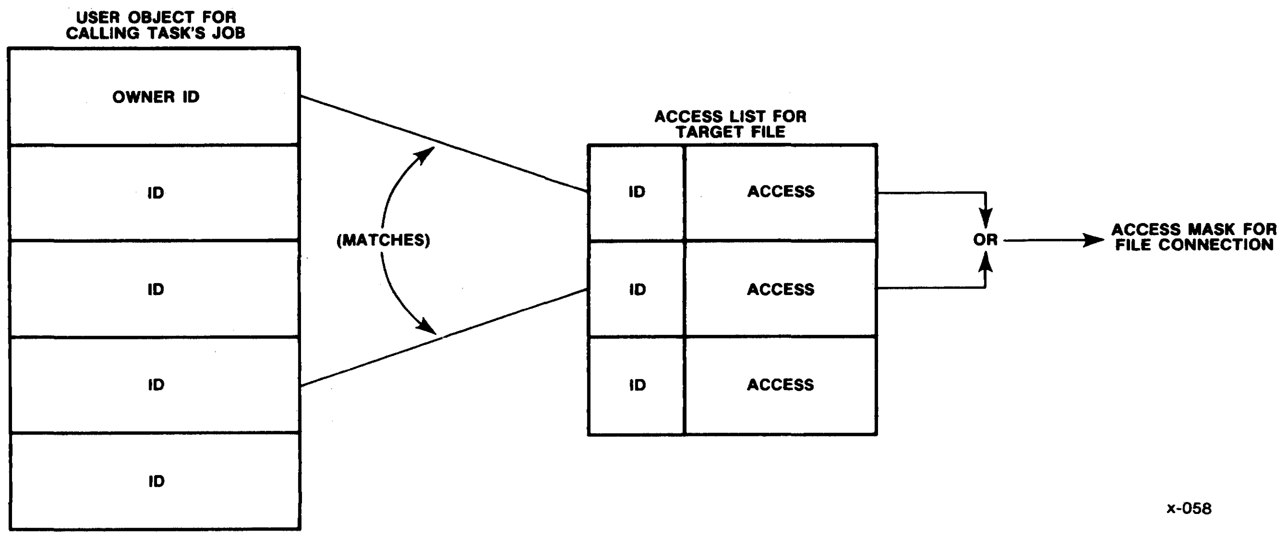

USER OBJECT FOR CALLING TASK'S JOB
OWNER ID
ID
ID
ID
ID
(MATCHES)
ACCESS LIST FOR TARGET FILE
ID
ACCESS
ID
ACCESS
ID
ACCESS
OR
ACCESS MASK FOR FILE CONNECTION
x-058

Figure 4-2. Computing the Access Mask for a File Connection

AN EXAMPLE

The following example helps you to understand how to use IDs, access masks, access lists, and user objects to permit each user in a system to have exactly the kinds of access that you want that user to have.

Referring back to Figure 4-1, suppose that Tom is to have all kinds of access to the file BATCH-1, that Bill is to have read and append access only, and that the members of Department 2 are to have read access only. Tom (or whoever creates BATCH-1) can arrange for these kinds of access by doing the following:

- Create three user objects, each with one ID number. Assume that the ID numbers are 4000H (for Tom), 8000H (for Bill), and F000H (for the people in Department 2.)

- Use A$CREATE$FILE to create the file BATCH-1. In the call to A$CREATE$FILE, use the token for the user object containing the 4000H ID number and specify the access mask 00001111B. This call returns a file connection that gives its user (Tom) all kinds of access to BATCH-1. At this point the access list for BATCH-1 has just one ID-access mask pair.

- Use A$CHANGE$ACCESS to add an ID-access mask pair to the access list of BATCH-1. The ID should be 8000H and the access mask should be 00000110B. This gives Bill read and append access to Batch-1. Now the access list for BATCH-1 has two ID-access mask pairs.

- Use A$CHANGE$ACCESS to add a third pair to the access list of BATCH-1. The ID should be F000H and the access mask should be 00000010B. This gives the people in Department 2 read access to BATCH-1.

- Inform Bill that he can read the contents of BATCH-1 and append new information to it. Describe to him the prefix and subpath that are needed to attach BATCH-1, and tell him to create a user object with the ID 8000H. Tell him to specify that user object when attaching BATCH-1.

- Inform the head of Department 2 that the members of that department can read the contents of BATCH-1. Describe for him the prefix and subpath needed to attach BATCH-1, and tell him to create a user object with the ID F000H. Tell him to specify that user object when attaching BATCH-1.

When Bill attaches BATCH-1, he receives a file connection that he can use in calls to A$READ. He also can use A$WRITE, provided that the file pointer for that connection is at the end of the file.

When the head of Department 2 attaches BATCH-1, he receives a file connection that can be used in calls to A$READ. After he gives the token for that connection to the members of his department, they can all use the connection to read BATCH-1.

Note that this example shows that one ID number can be used to give certain access rights to an individual and that another ID number can be used to give different access rights to a collection of individuals.

SPECIAL USERS

There are two ID numbers that can have special meaning to the Basic I/O System. One is the number 0 and the other is the number 0FFFFH.

If so indicated during the configuration process, the ID number 0 represents the "system manager." A user object containing this value is privileged in two respects. First, when it is used to create or attach a file, the resulting file connection automatically has read access to the file if it is a data file or list access to the file if it is a directory file. This is true even if the access list for the file does not contain an ID-access mask pair whose ID value is 0. The second privilege granted such a user object is that it can call A$CHANGE$ACCESS to change any file's access list.

The ID number 0FFFFH represents the "world." World is special in that any file whose owner ID is 0FFFFH can have its access list modified by anyone. That is, any file connection for that file can be used in a call to A$CHANGE$ACCESS.

EXTENSION DATA

For each named file on a random access volume, the Basic I/O System creates and maintains a file descriptor on the same volume. The first portion of the descriptor contains information for the Basic I/O System. The last portion, called extension data, is available to your operating system extension, unless you are using the Human Interface, in which case the first two bytes of the extension data are reserved. You specify the number (from 0 to 255, inclusive) of bytes of extension data for each named file on the volume, when formatting the volume with the FORMAT utility. The FORMAT utility is described in the iRMX 86 OPERATOR'S MANUAL.

If you are writing an operating system extension, and you want to record special information in a file's descriptor, you can use SET$EXTENSION$DATA to place the data into the trailing portion of the descriptor. GET$EXTENSION$DATA can be used to access this data when it is needed later.

NOTE

If you are using the Human Interface, you must take care not to destroy the data the Human Interface keeps in the first two bytes of file descriptors. To preserve this data, first use GET$EXTENSION$DATA to read and save the data. Next, modify as many as necessary of the remaining 255 bytes without disturbing the first two bytes. Finally, use SET$EXTENSION$DATA to transfer the data to the descriptor.

SYSTEM CALLS FOR NAMED FILES

Several system calls relate to iRMX 86 named files. Some of these calls are useful for both data and directory files, some for only 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 system calls. The descriptions are grouped by function rather than alphabetically. These descriptions are very brief. Chapter 8 of this manual contains detailed descriptions of the calls.

OBTAINING AND DELETING CONNECTIONS

Six system calls pertain to obtaining or 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.

- 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.

USER OBJECTS

Five system calls pertain directly to user objects. None of these calls are specifically related to data or directory files. The calls are:

- CREATE$USER

  This call is used to create a user object.

- DELETE$USER

  This call is used to delete a user object.

- INSPECT$USER

  This call is used to ascertain a user object's id and to find out to which groups the user belongs.

- SET$DEFAULT$USER

  Your application can use this call to establish a default user for any iRMX 86 job.

- GET$DEFAULT$USER

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

DEFAULT PREFIXES

Two calls pertain to default prefixes, and neither of these calls pertains 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

Six system calls allow you to manipulate the data in a file. All six can be used with data files, while only four apply to directory files. 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 A$GET$FILE$STATUS system call in Chapter 8. The remaining 14 bytes are the ASCII characters making up the name of the file to which the directory entry points. (A file's name is the last component of a path name.) The advantage in using the A$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$SEEK to position the file pointer at the location within the file to receive the information. The A$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$SEEK to position the file pointer at the first byte to be dropped. Then the application invokes the A$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 A$GET$FILE$STATUS and A$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$READ system call (which can also be used to read a data file) was discussed earlier, under the heading "Manipulating Data." The second system call is A$GET$DIRECTORY$ENTRY. This system call can be used only on directory files, and can be used without opening a connection.

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 application 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 A$GET$PATH$COMPONENT, and you can use it on data files and directories. See the description of this system call in Chapter 8 for an explanation of how you 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.

- A$SET$EXTENSION$DATA

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

- A$GET$EXTENSION$DATA

  This call provides a means of reading extension data. A$GET$EXTENSION$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

The system calls that can be used with named files 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 the 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.

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.

Five system calls pertain to user objects. Of the five, GET$DEFAULT$USER and CREATE$USER can be invoked at any time. Two others, DELETE$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, A$GET$FILE$STATUS and A$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 application has obtained a device connection.

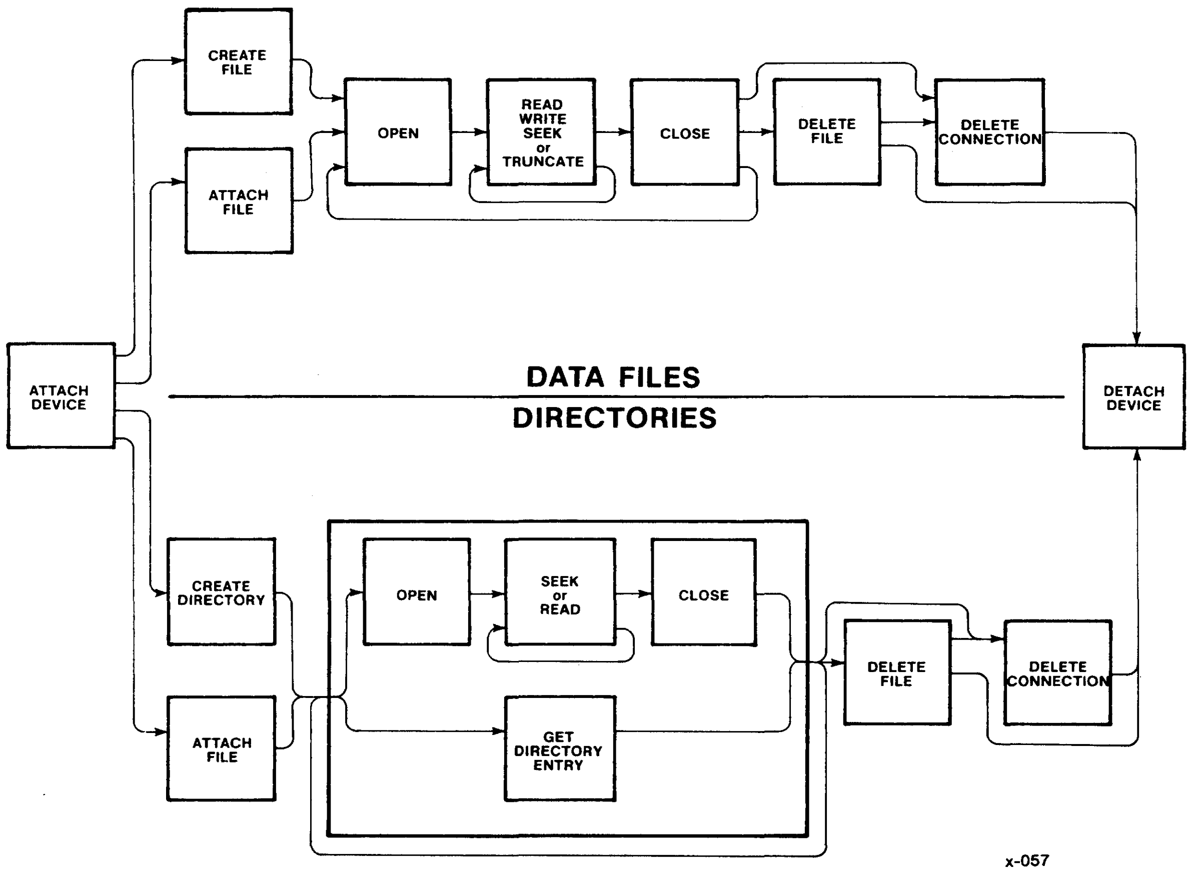


Figure 4-3. Chronology of Frequently Used System Calls for Named Files

CALLS RELATING TO EXTENSION DATA

The two system calls relating to extension data, A$GET$EXTENSION$DATA and A$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, A$GET$PATH$COMPONENT. Your application can use this call whenever the application has a connection to the file.

***

CHAPTER 5. 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. A physical file occupies an entire device, and the Basic I/O System provides your applications with the ability to capitalize on the physical characteristics 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. 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 than 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.

   Your application must call A$PHYSICAL$ATTACH$DEVICE to obtain a device connection for the device. This needs to be done only once for each device and 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 also that the file must be a physical file.

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 if the physical file has already been created. 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 can use either the device connection for the device or an existing file connection to the file as the PREFIX parameter in the system call.

   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.

4. Manipulate the file.

   Four system calls can be used to read, write, or otherwise manipulate your physical file:

   - The A$READ and A$WRITE system calls can be used to read from the device and write to the device, respectively.

   - 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.

- 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 file connection.

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

All of these system calls are described in Chapter 8.

***

CHAPTER 6. 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.

Because two tasks are involved in using each stream file, each task must perform one half of a protocol. There are several 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 call A$PHYSICAL$ATTACH$DEVICE to obtain a device connection before creating a stream file. This is necessary because, when your application invokes the A$CREATE$FILE system call, the device connection tells the Basic I/O System what kind of file to create.

   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 recommends the name "stream", but it 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 to 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 with the device connection to create the stream file and obtain a token for a file connection to the 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.

3. Pass the file connection to the reading task.

   There are several ways of doing this, including the use of 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 equal to the token for the file connection. Set the MODE parameter for writing. And set the SHARE parameter for 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.

All of these system calls are described in Chapter 8.

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 that of 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 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.

   This 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 equal to the token for the new connection. Set the MODE parameter for reading, and set the SHARE parameter for 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 writing task deletes the old connection, and, as soon as both connections have been deleted, the Basic I/O System deletes the stream file.

All of these system calls are described in Chapter 8.

***

CHAPTER 7. 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 7-1 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.

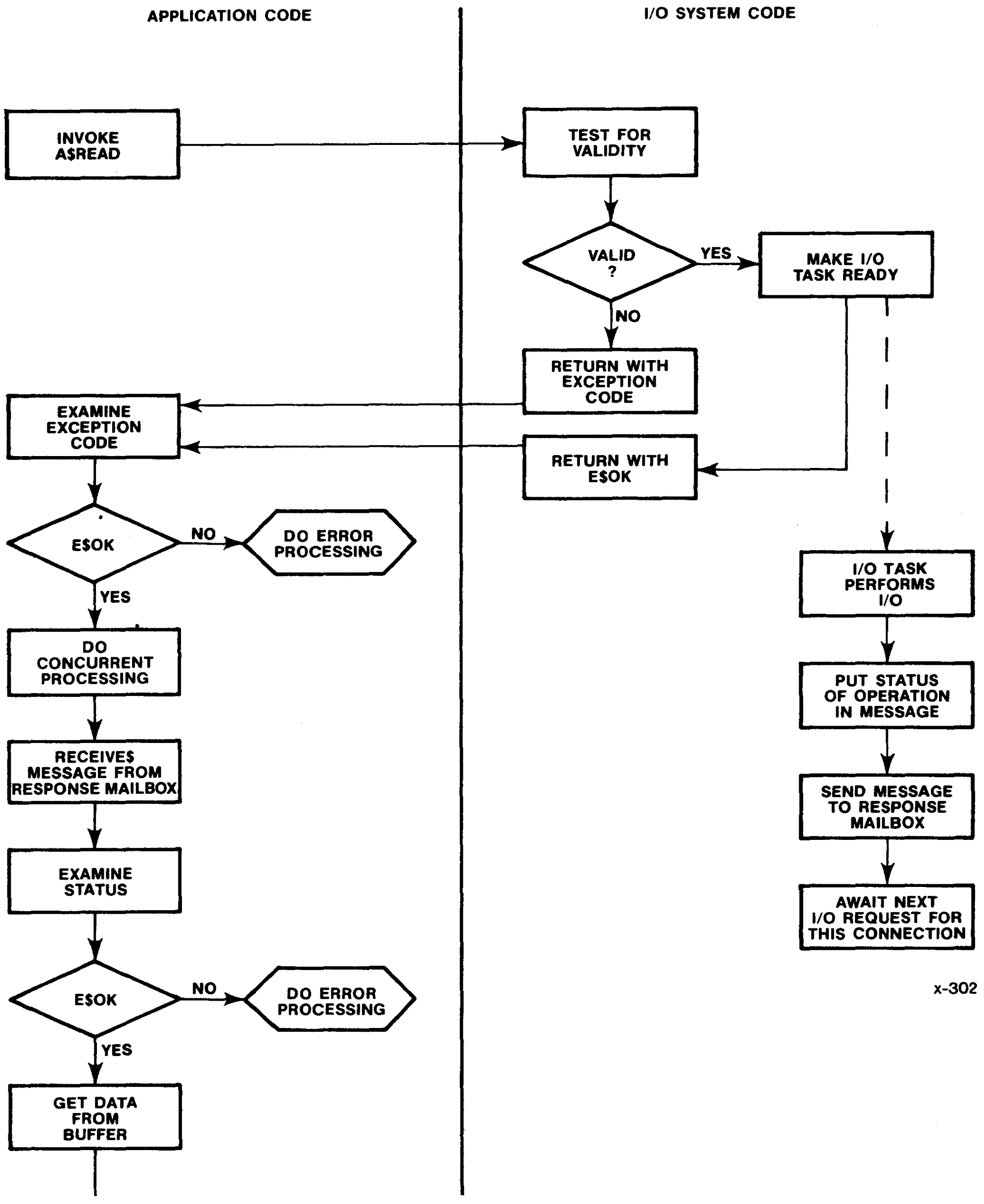


Figure 7-1. Concurrent Behavior of an Asynchronous System Call

4. Your application receives control. Its actions at this point depend 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 finish 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 condition code.

5. Your application now must have the information. Before taking the information from the buffer, your application must verify that the concurrent part of the A$READ system call ran successfully. There are two ways in which the task can do this. One way is for the application to issue a RECEIVE$MESSAGE system call to check the response mailbox that the application specified when it invoked the A$READ system call. The other way (which can be used only after a call to A$READ, A$WRITE, or A$SEEK) is for the application to issue a WAIT$IO system call, in which it passes a token for the response mailbox and receives the concurrent condition code directly.

   By using the RECEIVE$MESSAGE system call, the application obtains a segment that contains, among other things, a condition code for the concurrent part of the A$READ system call. If this condition 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 ascertain why the reading operation was not successful.

   By using the WAIT$IO system call, the application receives directly the condition code for the concurrent part of the A$READ system call. The application also receives directly another value. If the concurrent condition code for A$READ is E$OK, then this other value is the number of bytes successfully read; otherwise this other value has no significance.

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 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 allows your application to designate a response mailbox through which the application receives the result of the concurrent part of the system call.

- Whenever the sequential part of an asynchronous system call returns a condition code <u>other than</u> E$OK, your application should not attempt to receive a message from the response mailbox, nor should it call WAIT$IO. There can be no further information for the application 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$OK), your application can count on the Basic I/O System running the concurrent part of the system call. Your application can take advantage of the concurrency by doing some processing before receiving the message from the response mailbox or before calling WAIT$IO.

- 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. If more than one type of object can be returned, your application can ascertain the type of the returned object by calling GET$TYPE.

- Whenever the Basic I/O System returns a segment to your application's response mailbox and the application calls RECEIVE$MESSAGE to obtain information from that segment, the application should delete the segment when the segment is no longer needed. The Basic I/O System draws memory for such segments from the memory pool of the calling task's job, so if the application fails to delete such segments, the job might run short of memory.

- If your application calls WAIT$IO to obtain the results of a call to A$READ, A$WRITE, or A$SEEK, the application does not have access to the I/O result segment and therefore cannot delete it. While this seems to be a problem at first glance, it is actually an advantage, because it enables the Basic I/O System to maintain a supply of I/O result segments that it can use repeatedly. Because most I/O-related operations are reads, writes, or seeks, this means a significant performance enhancement for your application.

***

CHAPTER 8. SYSTEM CALLS

This chapter describes the PL/M-86 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-86 call. The precise format of each call is spelled out as part of its detailed description.

Basic I/O operations are declared as typed or untyped external procedures for PL/M-86. PL/M-86 programs perform I/O operations by making 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.

NOTE

A file connection that was obtained in one job cannot be used as a connection by another job. However, a file connection can be used as a prefix by other jobs in any call requiring prefix and subpath parameters. (The only exceptions to this rule are that the other jobs cannot use the connection as a prefix while specifying a null or zero-length subpath in calls to A$CHANGE$ACCESS or A$DELETE$FILE.) This means that a file connection can be passed to another job and the other job can obtain its own connection to the same file by calling A$ATTACH$FILE, with the passed file connection being used as the prefix parameter in the call.

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.
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

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 either call RECEIVE$MESSAGE and wait at the designated mailbox or call WAIT$IO. 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 -- A$ATTACH$FILE, A$CREATE$FILE, and A$PHYSICAL$ATTACH$DEVICE -- return to the mailbox a token for a connection if the system call performs successfully or an I/O result segment otherwise. After calling RECEIVE$MESSAGE to obtain the result of one of these system calls, a task should perform a GET$TYPE system call to ascertain the type of object returned to the response mailbox. The iRMX 86 NUCLEUS REFERENCE MANUAL describes the GET$TYPE system call in detail.

NOTE

I/O result segments should be deleted when they are no longer needed. Otherwise, they will consume available memory.

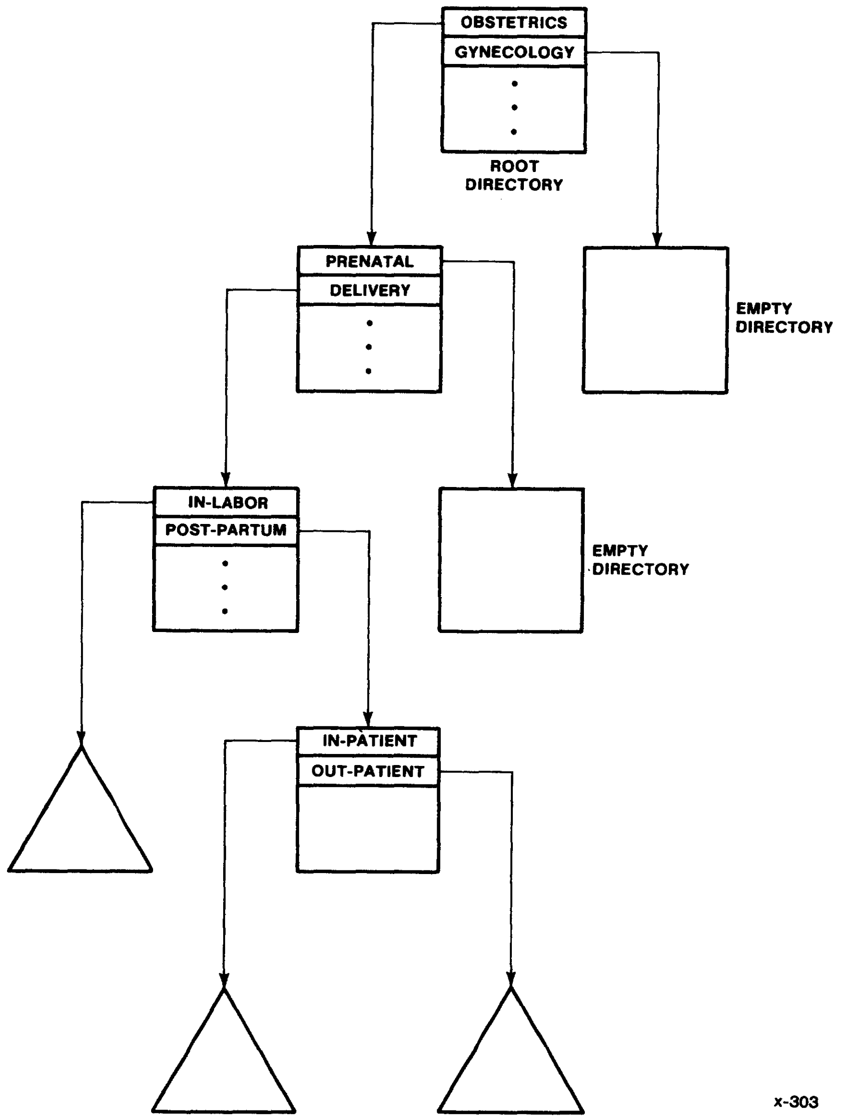


Figure 8-1. Sample Named File Tree

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 finishes 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.

CONDITION CODES

The Basic I/O System returns a condition 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 condition 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 a condition code for the sequential portion of the call to the word pointed to by the except$ptr parameter and a condition 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.

Throughout this chapter, PL/M-86 data types, such as BYTE and WORD, are used. In addition, the iRMX 86 data type TOKEN is used. Definitions of both PL/M-86 and iRMX 86 data types are given in Appendix A. Because TOKEN is not a PL/M-86 data type, if you use it you must declare it to be literally a WORD or a SELECTOR in every module in which it is used.

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.	8-122
GET$DEFAULT$PREFIX	Inspect default prefix.	8-115
SET$DEFAULT$USER	Set default user for job.	8-124
GET$DEFAULT$USER	Inspect default user.	8-117

DEVICE-LEVEL SYSTEM CALLS

System Call	Function	Page
A$PHYSICAL$ATTACH$-DEVICE	Asynchronous attach device.	8-66
A$PHYSICAL$DETACH$-DEVICE	Asynchronous detach device.	8-70
A$SPECIAL	Asynchronous perform device-level function.	8-87

FILE/CONNECTION-LEVEL SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$CREATE$FILE	Asynchronous data file creation.	*	*	*		8-28
A$ATTACH$FILE	Asynchronous attach file.	*	*	*	*	8-11
A$CREATE$DIRECTORY	Asynchronous directory file creation.				*	8-23
A$DELETE$CONNECTION	Asynchronous delete file connection.	*	*	*	*	8-34
A$DELETE$FILE	Asynchronous data or directory file deletion.		*	*	*	8-37

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.			*	*	8-15
A$RENAME$FILE	Asynchronous rename file.			*	*	8-76
A$TRUNCATE	Asynchronous truncate file.			*		8-101

FILE INPUT/OUTPUT SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$OPEN	Asynchronous open file.	*	*	*	*	8-62
A$SEEK	Asynchronous move file pointer.	*		*	*	8-81
A$READ	Asynchronous read file.	*	*	*	*	8-73
A$WRITE	Asynchronous write file.	*	*	*		8-107
A$CLOSE	Asynchronous close file.	*	*	*	*	8-20

FILE INPUT/OUTPUT SYSTEM CALLS (continued)

System Call	Function	P F	S F	N F	N D	Page
A$UPDATE	Asynchronous finish writing to output device.	*		*	*	8-104
WAIT$IO	Synchronous wait for status after I/O.	*	*	*	*	8-127

GET STATUS/ATTRIBUTE SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$GET$CON-NECTION$STATUS	Asynchronous get connection status.	*	*	*	*	8-42
A$GET$FILE$STATUS	Asynchronous get file status.	*	*	*	*	8-52
A$GET$DIRECTORY$ENTRY	Asynchronous inspect directory entry.				*	8-46
A$GET$PATH$COMPONENT	Asynchronous obtain path name from connection token.			*	*	8-58

USER OBJECT SYSTEM CALLS

System Call		Page
CREATE$USER	Create a user object.	8-111
INSPECT$USER	Get IDs in a user object.	8-120
DELETE$USER	Delete a user object.	8-113

EXTENSION DATA SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	Page
A$SET$EXTENSION$DATA	Asynchronous store a file's extension data.			*	*	8-84
A$GET$EXTENSION$DATA	Asynchronous receive a file's extension data.			*	*	8-49

TIME/DATE SYSTEM CALLS

System Call	Function	Page
SET$TIME	Set date/time value in internally-stored format.	8-126
GET$TIME	Get date/time value in internally-stored format.	8-119

A$ATTACH$FILE

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

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

INPUT PARAMETERS

user	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 TOKEN for the connection object to be used as the path prefix. A zero specifies the default prefix for the calling task's job.
subpath$ptr	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$mbox	A TOKEN into which the Basic I/O System places a token for the mailbox that receives the result object of the call. This result object is a new file connection if the call succeeds or an I/O result segment otherwise (see Appendix C). To ascertain the type of object returned, use the Nucleus system call GET$TYPE. If the object received is an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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. A$ATTACH$FILE has no effect on the owner ID or the access list for the file.

CONDITION CODES

A$ATTACH$FILE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a <u>sequential</u> condition code. A code returned as a result of asynchronous processing is a <u>concurrent</u> condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$DEV$OFF$LINE	The prefix parameter in this system call refers to a logical connection. One of the following is true: • The device has been physically attached but is now off-line. • The device has never been physically attached. (See Appendix E for a more detailed explanation.)
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The user parameter • The prefix parameter • The resp$mbox parameter 2. The prefix connection is being deleted.

E$LIMIT	Processing this call would cause one or more of these limits to be exceeded: • The maximum number (specified when the job was created) of objects allowed for this job. • The number (255 decimal) of I/O operations that can be outstanding at one time for the user object specified in the call. • The number (255 decimal) of I/O operations that can be outstanding at one time for the caller's job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NO$PREFIX	You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but a default prefix is not cataloged there.
E$NO$USER	If the user parameter in this call <u>is not zero</u>, then the problem is that the parameter is not a token for a user object. If the user parameter <u>is zero</u>, it specifies a default user. But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but a default user is not cataloged there. • The object that is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
E$NOT$CONFIGURED	A$ATTACH$FILE was not included when the system was configured.
E$PARAM	The specified path name contains invalid characters.

E$TYPE	One of more of the following conditions caused this exception: • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.) • The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The file specified is on a device that 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 that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)
E$IO	An I/O error occurred, and it might or might not have prevented the operation from being completed.
E$MEM	The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

A$CHANGE$ACCESS

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

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

INPUT PARAMETERS

user	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 TOKEN for the connection object to be used as the path prefix. A zero specifies the default prefix for the calling task's job.
subpath$ptr	A POINTER to a STRING giving the subpath of the file whose access is to be changed. A null string indicates that the prefix itself designates the desired file.
id	A WORD containing the ID number of the user whose access is to be changed. If this ID does not already exist in the ID-access mask 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. For each bit, a one grants access, and a zero denies it. (Bit 0 is the low-order bit.) 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)

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$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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 file's ID-access mask 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. However, if the owner is "WORLD", that is, OFFFFH, then any task may change the access mask of the file.

CONDITION CODES

A$CHANGE$ACCESS returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK

No exceptional conditions.

E$DEV$OFF$LINE

The prefix parameter in this system call refers to a logical connection. One of the following is true:

- The device has been physically attached but is off-line.
- 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. One or more of the following parameters is not a token for an existing object:
   - The user parameter
   - The prefix parameter
   - The response mailbox 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 would cause one or more of these limits to be exceeded:

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

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

E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NO$PREFIX	You specified a default prefix (prefix parameter equals zero). But no default prefix can be found because of one of the following: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but no default prefix is cataloged there.
E$NO$USER	If the user parameter in this call <u>is not zero</u>, then the problem is that the parameter is not a token for a user object. If the user parameter <u>is zero</u>, it specifies a default user. But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but no default user is cataloged there. • The object which is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
E$NOT$CONFIGURED	A$CHANGE$ACCESS was not included when the system was configured.
E$PARAM	The specified path name contains invalid characters.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One of more of the following conditions caused this exception:

- The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.)
- The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The file specified is on a device that the system is detaching.
E$FACCESS	The user object in the parameter list is not qualified for "change entry access" for the parent directory, nor is it the owner of the file.
E$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.
E$FTYPE	The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)
E$IO	An I/O error occurred, and it might or might not have prevented the operation from being completed.
E$MEM	The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$SUPPORT	The call attempted to add another access ID to the list of access ID's. The access list already contained 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 TOKEN for the file connection to be closed.

OUTPUT PARAMETERS

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

DESCRIPTION

The A$CLOSE system call closes an open file connection. It is called between uses of a file and when the application needs 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 response mailbox until the file is closed.

CONDITION CODES

A$CLOSE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a <u>sequential</u> condition code. A code returned as a result of asynchronous processing is a <u>concurrent</u> condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$LIMIT	Processing this call would have exceeded the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$CLOSE was not included when the system was configured.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One of more of the following conditions caused this exception: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The specified connection is not open.
E$IO	An I/O error occurred, but the operation was successful anyway.

A$CREATE$DIRECTORY

A$CREATE$DIRECTORY creates a directory file.

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

INPUT PARAMETERS

user	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 TOKEN for the connection to be used as the path prefix. A zero specifies the default prefix for the calling task's job.
subpath$ptr	A POINTER to a STRING containing the subpath of the directory to be created. The subpath string must not be null, and it 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$mbox	A TOKEN for the mailbox that receives the result object of this call. This result object is a directory file connection if the call succeeded, or an I/O result segment otherwise (see Appendix C). To ascertain the type of object returned, use the Nucleus system call GET$TYPE.

	If the object received is an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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.

CONDITION CODES

A$CREATE$DIRECTORY returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition 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 7 of this manual.

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

Sequential Condition Codes

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

E$OK — No exceptional conditions.

E$DEV$OFF$LINE — The prefix parameter in this system call refers to a logical connection. One of the following is true:

- The device has been physically attached but is now off-line.
- The device has never been physically attached. (See Appendix E for a more detailed explanation.)

E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: ● The user parameter ● The prefix parameter ● The response mailbox parameter 2. The prefix connection is being deleted.
E$IFDR	This system call applies only to named directory files, but the prefix and subpath parameters specify some other type of file.
E$LIMIT	Processing this call would cause one or more of these limits to be exceeded: ● The maximum number (specified when the job was created) of objects allowed for this job. ● The number (255 decimal) of I/O operations that can be outstanding at one time for the user object specified in the call. ● The number (255 decimal) of I/O operations that can be outstanding at one time for the caller's job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NO$PREFIX	You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one or more of the following reasons: ● When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix. ● The job's directory can have entries but no default prefix is cataloged there.
E$NO$USER	If the user parameter in this call <u>is not zero</u>, then the problem is that the parameter is not a user object. If the user parameter <u>is zero</u>, it specifies a default user. But no default user can be found because of one of the following reasons:

- When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user.
- The job's directory can have entries but no default user is cataloged there.
- The object that is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.

E$NOT$CONFIGURED — A$CREATE$DIRECTORY was not included when the system was configured.

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

E$TYPE — One or more of the following conditions caused this exception:

- The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.)
- The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK — No exceptional conditions.

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

E$FACCESS — The user object in the parameter list is not qualified for "add-entry" access to the parent directory.

E$FEXIST — A file with the specified path 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$FTYPE	The subpath parameter in the call contained a string which should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)
E$IO	An I/O error occurred, and it might or might not have prevented the operation from being completed.
E$MEM	The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$SPACE	One or more of the following is true: • The volume has no more space. • No more named files or directories can be created on this volume. The maximum number of files or directories that 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.)

A$CREATE$FILE

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

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

INPUT PARAMETERS

user	A TOKEN for the user object of the owner of the new file. It also furnishes the user ID for any access checking that might occur. A zero specifies the default user for the calling task's job. This parameter does not apply to physical or stream files.
prefix	A TOKEN for a device or file connection. The file created by this call is of the type (physical, stream, or named) that is associated with this parameter. A zero for this parameter specifies the default prefix for the job. 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.
subpath$ptr	A POINTER to a STRING containing the subpath for the named file being created. This parameter does not apply to physical and stream files.
access	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. (Bit 0 is the low-order bit.)

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

This parameter does not apply to physical or stream files.

granularity	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 become noncontiguous when it is extended.

The granularity parameter can have the following values:

0	Same as volume granularity
FFFF	The file must be contiguous
Other	Number of bytes per 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.

size	A DWORD giving the number of bytes initially reserved for the file. For stream files, this value must equal zero. For physical files, this parameter is ignored.
must$create	A BYTE whose value (OFFH for TRUE or 0 for FALSE) determines the handling of input paths designating an existing file (see following DESCRIPTION).

OUTPUT PARAMETERS

resp$mbox	A TOKEN for the mailbox that receives the result object of this call. This result object is a new file connection if the call succeeded or an I/O result segment otherwise (see Appendix C). To ascertain the type of object returned, use the Nucleus system call GET$TYPE.

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

except$ptr	A POINTER to a WORD where the sequential condition 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 occurs:

- Error: If the "must$create" parameter is TRUE (OFFH), an error condition code (E$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$FILE). In this case, the file is truncated or expanded according to the "size" parameter, so data in the file might be lost. As in the case of A$ATTACH$FILE, the file's owner ID and access list are unchanged.

- 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. Because 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.

CONDITION CODES

A$CREATE$FILE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition 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 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$DEV$OFF$LINE	The prefix parameter in this system call refers to a logical connection. One of the following is true: • The device has been physically attached but is now off-line. • The device has never been physically attached. (See Appendix E for a more detailed explanation.)
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The user parameter • The prefix parameter • The resp$mbox parameter 2. The prefix connection is being deleted.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NO$PREFIX	You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but a default prefix is not cataloged there.
E$NO$USER	If the user parameter in this call is not zero, then the problem is that the parameter is not a token for a user object. If the user parameter is zero, it specifies a default user. But no default user can be found because of one of the following reasons:

	• When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but a default user is not cataloged there. • The object which is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
E$NOT$CONFIGURED	A$CREATE$FILE was not included when the system was configured.
E$PARAM	The specified path name contains invalid characters.
E$TYPE	One or more of the following conditions caused this exception: • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.) • The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The file specified is on a device that 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$FTYPE	The subpath parameter in the call contained a string that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed. Try the operation again. If E$IO is returned again, then the operation is not being performed successfully.
E$MEM	The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$SHARE	The file this call is attempting to create already exists and is open. It was opened with the characteristic "no share with writers." (See the A$OPEN call in this chapter.)
E$SPACE	one or more of the following is true: • The volume has no more space. • No more named files or directories can be created on this volume. The maximum number of files or directories that can be created on a particular volume is set when that volume is formatted. (See the description of the FORMAT Command in the iRMX 86 HUMAN INTERFACE REFERENCE MANUAL.)
E$SUPPORT	The file exists, and the must$create parameter is FALSE. When the Basic I/O System was configured, an option was chosen that prevented this combination, so that files could not be automatically truncated to zero size. See the DESCRIPTION section.

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 TOKEN for the file connection to be deleted.

OUTPUT PARAMETERS

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

DESCRIPTION

The A$DELETE$CONNECTION system call deletes a connection object. It also deletes the associated file if both of the following are true:

- The file is already marked for deletion (by a previous A$DELETE$FILE call)
- The specified connection is the only 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.

CONDITION CODES

A$DELETE$CONNECTION returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection parameter is a device connection, not a file connection.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$LIMIT	Processing this call would exceed the number (specified during configuration) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$DELETE$CONNECTIION was not included when the system was configured.
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.

E$TYPE	One or more of the following is a token for an object that is not of the correct type: • The connection parameter. • The resp$mbox parameter.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$IO	An I/O error occurred, but the operation was performed successfully anyway.

A$DELETE$FILE

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

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

INPUT PARAMETERS

user	A TOKEN for the user object to be inspected in access checking. A zero specifies the default user for the calling task's job. This parameter does not apply to physical or stream files.
prefix	A TOKEN for the connection object to be used as the path prefix. A zero specifies the default prefix for the calling task's job.
subpath$ptr	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. This parameter does not apply to physical or stream files.

OUTPUT PARAMETERS

resp$mbox	A TOKEN for a mailbox that receives an I/O result segment (see Appendix C) when the file is marked for deletion. The file will not actually be deleted until all connections to the file are deleted, as explained under the DESCRIPTION below. A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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.

CONDITION CODES

A$DELETE$FILE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK — No exceptional conditions.

E$DEV$OFF$LINE — The prefix parameter in this system call refers to a logical connection. One of the following is true:

- The device has been physically attached but is now off-line.
- 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. One or more of the following parameters is not a token for an existing object:

   - The user parameter
   - The prefix parameter
   - The response mailbox parameter

2. The prefix connection is being deleted.

E$IFDR	This system call applies only to named or stream files, but the prefix and subpath parameters specified a physical file.
E$LIMIT	Processing this call would cause one or more of these limits to be exceeded: • The maximum number (specified when the job was created) of objects allowed for this job. • The number (255 decimal) of I/O operations that can be outstanding at one time for the user object specified in the call. • The number (255 decimal) of I/O operations that can be outstanding at one time for the caller's job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NO$PREFIX	You specified a default prefix (prefix argument equals zero). But no default prefix can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but no default prefix is cataloged there.
E$NO$USER	If the user parameter in this call is not zero, then the problem is that the parameter is not a token for a user object. If the user parameter is zero, it specifies a default user. But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user.

	• The job's directory can have entries but no default user is cataloged there.
	• The object that is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
E$NOT$CONFIGURED	A$DELETE$FILE was not included when the system was configured.
E$PARAM	The specified path name contains invalid characters.
E$SUPPORT	The specified connection is not valid in this system call because it was not created by this job.
E$TYPE	One or more of the following conditions caused this exception:
	• The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.)
	• The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	One of the following caused this exception:
	• The file specified is on a device that the system is detaching.
	• The call is attempting to delete a stream file that is already marked for deletion.
	• The call is attempting to delete a directory containing entries.
E$FACCESS	One of the following caused this exception:
	• The user object specified by the prefix and subpath parameters does not have delete access to this file.
	• The call attempted to delete the root directory or a bit map 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 that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.
E$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.

A$GET$CONNECTION$STATUS

A$GET$CONNECTION$STATUS returns information about a file connection.

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

INPUT PARAMETER

connection	A TOKEN for the file connection whose status is desired.

OUTPUT PARAMETERS

resp$mbox — A TOKEN for the mailbox that is to receive a connection-status segment. The calling task is responsible for deleting the connection-status segment after examining it.

The information in this segment is structured as follows:

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

These fields are interpreted as follows:

status — A condition code giving the outcome of the status-fetch operation. 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
2	Stream
4	Named

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. (Bit 0 is the low-order bit.)

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$mode — The sharing mode 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

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

access — The access rights for this connection. For each bit, a one grants access and a zero denies it. (Bit 0 is the low-order bit.)

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 condition code will be returned.

DESCRIPTION

The A$GET$CONNECTION$STATUS system call returns a segment containing status information about a file connection.

CONDITION CODES

A$GET$CONNECTION$STATUS returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$LIMIT	Processing this call would exceed the number (specified during configuration) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$GET$CONNECTION$STATUS was not included when the system was configured.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One of or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System returns one of the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. You are responsible for deleting this segment.

E$OK	No exceptional conditions.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.

A$GET$DIRECTORY$ENTRY

A$GET$DIRECTORY$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 TOKEN for the directory file with the desired entry.
entry$num	A WORD giving the entry number of the desired file name. Entries within a directory are numbered sequentially starting from zero. The E$EMPTY$ENTRY condition code will be returned if there is no entry associated with this number.

OUTPUT PARAMETERS

resp$mbox — A TOKEN for the mailbox that will receive a directory-entry segment. The task making the A$GET$DIRECTORY$ENTRY call is responsible for deleting this segment after examining it.

Information in this segment is structured as follows:

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

where:

status	Indicates how the operation was completed. E$OK, E$EMPTY$ENTRY, and E$DIR$END condition codes all indicate successful completion.
name	File name contained in the specified entry, padded with blanks. This field is valid only if status = E$OK.

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

DESCRIPTION

The A$GET$DIRECTORY$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.

CONDITION CODES

A$GET$DIRECTORY$ENTRY returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a <u>sequential</u> condition code. A code returned as a result of asynchronous processing is a <u>concurrent</u> condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$IFDR	This system call applies only to named directories, but the connection parameter specifies another type of file.
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.

E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$GET$DIRECTORY$ENTRY was not included when the system was configured.
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 more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$DIR$END	The entry$num parameter is greater than the number of entries in the directory.
E$EMPTY$ENTRY	The file entry designated in the call is empty.
E$FACCESS	The specified connection is not qualified for "display" access to the directory.
E$FTYPE	The specified connection does not refer to a directory.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.

A$GET$EXTENSION$DATA

The A$GET$EXTENSION$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);
```

INPUT PARAMETERS

connection	A TOKEN of a connection to a file whose extension data is desired.
resp$mbox	A TOKEN for the mailbox that will receive a segment containing the named file-status information. The calling task is responsible for deleting this segment after examining it.

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

```
      DECLARE ext$data$seg STRUCTURE (
        status    WORD,
        count     BYTE,
        info(*)   BYTE);
```

These fields are interpreted as follows:

status	A condition code indicating the outcome of the status-fetch operation. If this code is not E$OK, the remaining fields must be considered invalid.
count	A number (from 0 to 255 decimal) indicating the number of bytes returned.
info	The extension data.

OUTPUT PARAMETER

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

DESCRIPTION

Associated with each file created through the Basic I/O System is a file descriptor containing information about the file. Some of that information is used by the Basic I/O System and can be accessed by tasks through the A$GET$FILE$STATUS system call. Up to 255 additional bytes of the file descriptor, known as extension data, are available for use by Operating System extensions. OS extensions can write extension data by using A$SET$EXTENSION$DATA and they can read extension data by using A$GET$EXTENSION$DATA.

When a task calls A$GET$EXTENSION$DATA, it specifies a response mailbox to which the system returns a segment with the extension data. The information is located in the low-memory portion of the segment. A$GET$EXTENSION$DATA can only be applied to connections created via the named file driver.

CONDITION CODES

A$GET$EXTENSION$DATA can return condition 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 7 of this manual.

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

Sequential Condition 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$OK	No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter. 2. The connection is being deleted.
E$IFDR	This system call applies only to named files, but the prefix and subpath parameters specify another type of file.

E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$GET$EXTENSION$DATA was not included when the system was configured.
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Exception Codes

The Basic I/O System will return one of 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$OK	No exceptional conditions.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.

A$GET$FILE$STATUS

A$GET$FILE$STATUS returns status and attribute information about a file.

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

INPUT PARAMETER

connection	A TOKEN for a connection to the file whose status is sought.

OUTPUT PARAMETERS

resp$mbox — A TOKEN for the mailbox that receives a segment containing a data structure with the status information for the specified file. The information in the first part of this structure -- down to the dev$conn field -- is returned for any file (physical, stream, or named), but information from the file$id field down to the end of the structure is provided only for named files. The contents of the named$file field indicates whether the file is a named file.

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

OUTPUT PARAMETERS

resp$mbox (continued)

Information from this point on is returned only if the file is a named file.

```
                                   file$id         WORD,
                                   file$type       BYTE,
                                   file$gran       BYTE,
                                   owner$id        WORD,
                                   create$time    DWORD,
                                   access$time    DWORD,
                                   modify$time    DWORD,
                                   file$size      DWORD,
                                   file$blocks    DWORD,
                                   vol$name(6)     BYTE,
                                   vol$gran        WORD,
                                   vol$size       DWORD,
                                   accessor$count  WORD,
                                   first$access    BYTE,
                                   first$ID        WORD,
                                   second$access   BYTE,
                                   second$ID       WORD,
                                   third$access    BYTE,
                                   third$ID        WORD);
```

These fields are interpreted as follows:

status	A condition code indicating how the get file status operation was completed. If this code is not E$OK, the remaining fields must be considered invalid.
num$conn	The number of connections to the file.
num$reader	The number of connections currently open for reading.
num$writer	The number of connections currently open for writing.
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$file	Tells whether this structure contains any information beyond the dev$conn field. OFFH means yes and 0 means no.

dev$name	The name of the physical device where this file resides. This name is padded with blanks. To ensure the uniqueness of file names, they should not be more than 13 characters in length.
file$drivers	A bit map that tells what kinds of files can reside on this device. If bit n is on, then file driver n+1 can be used. Bit 0 is the low-order bit.

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

functs	A bit map that describes the functions supported by the device where this file resides. A bit set to one indicates the corresponding function is supported. Bit 0 is the low-order bit.

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

flags	Meaningful only for diskette drives. This field is interpreted as follows. (Bit 0 is the low-order bit.)

Bit	Meaning
0	0=bits 1-7 are not significant 1=bits 1-7 are significant
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	0=standard diskette, meaning that track 0 is single-density with 128-byte sectors 1=a non-standard diskette or not a diskette
5-7	reserved

dev$gran	The device granularity, in bytes, of the device where this file resides.
dev$size	The storage capacity of the device, in bytes.
dev$conn	The number of connections to the device.

The information from here to the end of the structure is returned only for named files, as indicated by a value of OFFH in the named$file field.

file$id	A number that distinguishes this file from all other files on the same device.
file$type	Indicates the type of the file: 6 means directory file; and 8 means data file.
file$gran	The file granularity, as a multiple of vol$gran. For example, if file$gran is 2 and vol$gran is 256, then the file's granularity is 512.
owner$id	The first ID in the user object that was presented to the Basic I/O System when the file was created.
create$time	The time and date when the file was created. Whether the Basic I/O System maintains this field is a configuration option.
access$time	The time and date when the file was last accessed. Whether the Basic I/O System maintains this field is a configuration option.
modify$time	The time and date when the file was last modified. Whether the Basic I/O System maintains this field is a configuration option.
file$size	The total size of the file, in bytes.
file$blocks	The number of volume blocks allocated to this file. A volume block is a contiguous area of storage that contains vol$gran bytes of data.
vol$name	The left-adjusted, null-padded ASCII name for the volume containing this file.
vol$gran	The volume granularity, in bytes.
vol$size	The storage capacity, in bytes, of the volume on which this file is stored.

accessor$count — The number of IDs in the user object that was presented to the Basic I/O System when this file was created.

first$access
second$access
third$access — Access masks for as many ID's as are indicated by id$count. The bits of the access masks are defined in the following table. An access right is granted if the appropriate bit is set to 1; otherwise, that right is denied. Bit 0 is the low-order bit.

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 condition code will be returned.

DESCRIPTION

The A$GET$FILE$STATUS system call returns status and attribute information about the designated file. Certain 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.

CONDITION CODES

A$GET$FILE$STATUS returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$GET$FILE$STATUS was not included when the system was configured.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following parameters is a token for an object of the wrong type: • The connection parameter • The resp$mbox parameter

Concurrent Condition Code

The Basic I/O System returns one of the following condition code in an I/O result segment at the mailbox specified by resp$mbox. You are responsible for deleting this segment.

E$OK	No exceptional conditions.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.

A$GET$PATH$COMPONENT

A$GET$PATH$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 TOKEN for the file connection whose name is sought.

OUTPUT PARAMETERS

resp$mbox — A TOKEN for the mailbox that will receive the file$name segment. This segment contains the file name associated with the designated connection and is structured as follows:

```
DECLARE  file$name  STRUCTURE(
         status     WORD,
         name       STRING);
```

These fields are interpreted as follows:

status	A condition code indicating the outcome of the operation.
name	A 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.

The task that makes the A$GET$PATH$COMPONENT call is responsible for deleting the file$name segment after examining it.

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

DESCRIPTION

A caller who knows the token for a connection to a file can invoke this system call and receive the name of the file in return. This name 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.

A$GET$PATH$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.

The A$GET$PATH$COMPONENT system call can be used in combination with the A$ATTACH$FILE system call to derive all of the components of a path name. Suppose, for example, that a file has the path name A/B/C, and that your task has only a token for the file. The following sequence of calls will reveal all of the path's components:

1. Call A$GET$PATH$COMPONENT to obtain the file name C.

2. Call A$ATTACH$FILE with the prefix parameter equal to the token for file C and the subpath equal to a circumflex (^). This call will return a token for a connection to directory file B.

3. After calling GET$TYPE to verify that the token is indeed for a connection, call A$GET$PATH$COMPONENT to obtain the file name B.

4. Call A$ATTACH$FILE with the prefix parameter equal to the token for file B and the subpath equal to a circumflex (^). This call will return a token for a connection to directory file A.

5. After calling GET$TYPE to verify that the token is indeed for a connection, call A$GET$PATH$COMPONENT to obtain the file name A.

6. Call A$ATTACH$FILE with the prefix parameter equal to the token for file A and the subpath equal to a circumflex (^). This call will return a token for a connection to the root of the file tree.

7. After calling GET$TYPE to verify that the token is indeed for a connection, call A$GET$PATH$COMPONENT again. This time, the null string will be returned, and this tells you that you now have all of the components of the desired path name.

CONDITION CODES

A$GET$PATH$COMPONENT returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 4 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$GET$PATH$COMPONENT was not included when the system was configured:
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$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, which might or might not have prevented the operation from being completed.
E$MEM	The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

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 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$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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.

The mode and share parameters are 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, because 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.

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 readers that subsequently read the modified part of the file.

NOTE

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

CONDITION CODES

A$OPEN returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.

E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$OPEN was not included when the system was configured.
E$PARAM	The mode or share parameter has an invalid value (out of range 1-3 or 0-3, respectively.)
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection is a file or directory connection that 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$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.

E$SHARE	One or more of the following conditions caused this exception: • The current file share characteristic is not compatible with the mode or the share parameter in the A$OPEN call. • This A$OPEN call is an attempt to open a directory for some operation other than "read" (mode parameter) and "share with all users" (share parameter). (See DESCRIPTION for more information on sharing files.)

A$PHYSICAL$ATTACH$DEVICE

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

CAUTION

Any task that uses this system call loses its device independence. To maintain as much device independence as possible in your application, a few selected tasks should perform all attaching and detaching of devices, passing tokens for the devices to other tasks as necessary.

Also, if a task uses this system call to attach devices, the devices are automatically detached (and connections to files on the device are automatically deleted) when the containing job is deleted. If you want to prevent the device from being detached in these circumstances, use the Extended I/O System's LOGICAL$ATTACH$DEVICE system call instead.

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

INPUT PARAMETERS

dev$name	A POINTER to a STRING containing the name (as specified during configuration) of the device to be attached. To prevent possible duplication of names, this name should be no more than 13 characters in length.
file$driver	A BYTE specifying which file driver is to supply the connection to the device. Possible values are as follows:

Value	File Driver
1	Physical
2	Stream
4	Named

resp$mbox	A TOKEN for the mailbox that receives the result object of the call. This result object is a new connection if the call is successful, or an I/O result segment otherwise (see Appendix C). To ascertain the type of object returned, use the Nucleus system call GET$TYPE. If the object received is an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.

OUTPUT PARAMETER

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

DESCRIPTION

A$PHYSICAL$ATTACH$DEVICE returns a device connection to the device specified by the dev$name parameter. The file$driver parameter specifies the kind of files (physical, stream, or named that the device will create when the returned device connection is used in subsequent calls to A$CREATE$FILE.

The device connection object is returned to the response mailbox if the call is successful; otherwise an I/O result segment is returned to the response mailbox. The returned connection object can be used as a prefix in other system calls. It can be deleted only by calling A$PHYSICAL$DETACH$DEVICE.

In the case of a connection to a disk device, where the file$driver parameter specifies named files for the device, the connection is actually to a volume mounted on the disk hardware. Such volumes must be properly formatted. If they are not, an E$ILLVOL exceptional condition is returned. Refer to the iRMX 86 OPERATOR'S MANUAL for information about formatting disks.

CONDITION CODES

A$PHYSICAL$ATTACH$DEVICE can return condition 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 7.

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

Sequential Condition 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$OK	No exceptional conditions.
E$EXIST	The resp$mbox parameter does not refer to an existing object.
E$LIMIT	Processing this call would cause one or more of the following limits to be exceeded: • The maximum number (specified when the job was created) of objects allowed for this job. • The number (255 decimal) of I/O operations that can be outstanding at one time for the caller's job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$PARAM	One or more of the following is true: • The number representing the file driver is not valid. • A value of zero was specified for the response mailbox.
E$TYPE	The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Exception Codes

The Basic I/O System will 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$CONTEXT	The specified device is already attached.
E$DEVFD	The specified device is not compatible with the specified file driver.
E$FNEXIST	The device specified by the device$name parameter does not exist.

E$ILLVOL	One or more of the following is true: • The specified device is a disk volume not properly formatted for use with the named file driver. The volume being attached must have FD$NAMED in its label. • The connection object returned to the response mailbox is a connection to the root directory of the attached device, and the fnode of this root directory is invalid.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.
E$MEM	The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

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);
```

INPUT PARAMETERS

connection	A TOKEN for the connection object for the device that is to be detached.
hard	A BYTE containing a value that specifies whether (OFFH) or not (0) a hard detach of the device is desired.
resp$mbox	A WORD containing a TOKEN for the mailbox to which the result segment is sent when the operation has finished. A value of zero indicates that no response is desired.

OUTPUT PARAMETER

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

DESCRIPTION

The A$PHYSICAL$DETACH$DEVICE system call breaks connections established by calls to A$PHYSICAL$ATTACH$DEVICE. It also deletes the file connection objects associated with those device connections. Devices that are detached in this manner must be reattached before any files on the device can be attached or reattached.

When detaching a device, you can choose to detach all attached files on the device. A hard detach deletes the connection objects for all such files on the device. To specify a hard detach, assign the value OFFH to the hard parameter.

If you choose not to request a hard detach, there must not be any attached files on the device. To specify that you do not want a hard detach, assign the value 0 to the hard parameter.

Note that, whether you specify a hard detach or not, there will be no attached files on the device after the device is detached.

CONDITION CODES

A$PHYSICAL$DETACH$DEVICE can return condition 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 7 of this manual.

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

Sequential Condition 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$OK	No exceptional conditions.
E$CONTEXT	The specified connection parameter is not a device connection.
E$EXIST	One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$PHYSICAL$DETACH$DEVICE was not included when the system was configured.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.

E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Exception Codes

The Basic I/O System will 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$CONTEXT	A soft detach was specified when connections to the device still existed.
E$FNEXIST	The device specified by the connection parameter is being detached.
E$IO	An I/O error occurred during the operation, but the operation was successful anyway.

A$READ

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

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

INPUT PARAMETERS

connection	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 TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (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 call DELETE$SEGMENT to delete the segment after examining it.

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 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 condition code will be returned.

DESCRIPTION

The A$READ system call initiates a read operation on an open connection. The data is read as a string of bytes, starting at the current location of the connection's 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).

CONDITION CODES

A$READ returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection parameter is a connection produced by the Extended I/O System.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.

E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$READ was not included when the system was configured.
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	This connection is not open for reading or updating.
E$FLUSHING	The connection was closed before the read operation was completed.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.
E$SPACE	The read operation attempted to read past the end of the physical device. This applies only to physical files.

A$RENAME$FILE

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

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

INPUT PARAMETERS

connection	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 TOKEN for the user object to be inspected in access checking. A zero specifies the default user for the job.
prefix	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$ptr	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$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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 concerning the manner in which a directory can be renamed. Any attempt to rename a directory as its own parent causes 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

In order to rename a file, the caller must have delete access to the file and must have add-entry access to the file's parent directory.

CONDITION CODES

A$RENAME$FILE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a <u>sequential</u> condition code. A code returned as a result of asynchronous processing is a <u>concurrent</u> exception code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$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.

E$DEV$OFF$LINE

The prefix parameter in this system call refers to a logical connection. One of the following is true:

- The device has been physically attached but is now off-line.
- 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. One or more of the following parameters is not a token for an existing object:
   - The connection parameter
   - The user parameter
   - The prefix parameter
   - The resp$mbox parameter
2. One or more of the following is being deleted:
   - The connection specified by the prefix.
   - The connection specified by the connection parameter.

E$IFDR

This system call applies only to named files, but the connection parameter specifies some other type of file.

E$LIMIT

Processing this call would cause one or more of the following limits to be exceeded:

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

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.

E$NO$PREFIX	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. ● The job's directory can have entries but a default prefix is not cataloged there.
E$NO$USER	If the user parameter in this call <u>is not zero</u>, then the problem is that the parameter is not a user object. If the user parameter <u>is zero</u>, it specifies a default user object. But no default user object 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 object. ● The job's directory can have entries but a default user object is not cataloged there. ● The object which is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
E$NOT$CONFIGURED	A$RENAME$FILE was not included when the system was configured.
E$PARAM	The path name contains invalid characters, or has a length of zero.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: ● The connection parameter is a token for an object that is not a connection object. ● The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object (Logical device objects are created by the Extended I/O System.) ● The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	One or more of the following is true: • The file specified is on a device that the system is detaching. • The call is attempting to rename the directory to a new path containing itself. This is specifically forbidden; see DESCRIPTION.
E$FACCESS	One or more of the following is true: • The specified connection does not have "add entry" access to the parent directory. • The specified user does not have "delete" access to the file. • The call is attempting to rename the root directory or a bit-map file.
E$FEXIST	A file with the specified path name already exists.
E$FNEXIST	One or more of the following is true: • A file in the specified path does not exist. • A file in the specified path is marked for deletion.
E$FTYPE	The subpath parameter in the call contained a file that should have been a directory, but is not. (Except for the last file, each file listed in a pathname must be a named directory.)
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.
E$MEM	The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$SPACE	There is no more space on this volume.
E$SUPPORT	As configured, 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, move$size, resp$mbox, except$ptr);
```

INPUT PARAMETERS

connection	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 move$size bytes; if this action moves the pointer past the beginning of the file, the pointer is set to zero (first byte). 2 Set the pointer to the location specified by move$size. 3 Move the file pointer forward by move$size bytes. 4 Move the pointer to the end of the file, minus move$size bytes.
move$size	A DWORD giving the number of bytes involved in the seek. The interpretation of move$size depends on the mode setting, as just explained.

OUTPUT PARAMETERS

resp$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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, 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.

CONDITION CODES

A$SEEK returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection parameter was produced by the Extended I/O System.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox 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 (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.

E$NOT$CONFIGURED	A$SEEK was not included when the system was configured.
E$PARAM	The mode parameter value is out of the valid range (1 to 4).
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$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, which might or might not have prevented the operation from being completed.
E$PARAM	This call attempted to seek beyond the end of the physical device. This applies only to physical files.

A$SET$EXTENSION$DATA

The A$SET$EXTENSION$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);
```

INPUT PARAMETERS

connection — A TOKEN for an asynchronous connection to a file whose extension data is to be set.

data$ptr — A POINTER to a structure of the following form:

```
DECLARE ext$data$seg  STRUCTURE(
              count      BYTE,
              info(*)    BYTE);
```

where:

count	Number (up to 255) of bytes of extension data being written.
info(*)	The extension data.

resp$mbox — A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (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 call DELETE$SEGMENT to delete the segment after examining it.

OUTPUT PARAMETER

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

DESCRIPTION

Associated with each file created through the Basic I/O System is a file descriptor containing information about the file. Some of that information is used by the Basic I/O System and can be accessed by tasks through the A$GET$FILE$STATUS system call. Up to 255 additional bytes of the file descriptor, known as extension data, are available for use by Operating System extensions, depending upon how the volumes were formatted. OS extensions can write extension data by using A$SET$EXTENSION$DATA and they can read extension data by using A$GET$EXTENSION$DATA. The maximum number of bytes of extension data may be less than 255 since the limit is specified when the secondary storage devices are formatted.

NOTE

The Human Interface uses the first two bytes of extension data. If your application includes the Human Interface, take care, when using A$SET$EXTENSION$DATA, to preserve the first two bytes. Do this by calling A$GET$EXTENSION$DATA before writing into the remaining bytes.

After the new extension data is set, an I/O result segment returns to the response mailbox.

A$SET$EXTENSION$DATA can only be applied to asynchronous connections created via the named file driver.

CONDITION CODES

A$SET$EXTENSION$DATA returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition 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	No exceptional conditions.

E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter. • The resp$mbox parameter. 2. The connection is being deleted.
E$IFDR	This system call applies only to named files, but the parameter list specified another type of file.
E$LIMIT	The call cannot be processed without exceeding the maximum number of objects allowed for this job (specified when the job was created).
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$SET$EXTENSION$DATA was not included when the system was configured.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Exception Codes

The Basic I/O System will 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$OK	No exceptional conditions.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.
E$PARAM	The count field in the ext$data$seg data structure contains a value greater than the value specified when the disk was formatted.

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 TOKEN for a connection to the file for which 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
stream	0	query
stream	1	satisfy
physical or named	2	notify
physical	3	get disk data
physical	4	get terminal data
physical	5	set terminal data
physical	6	set signal

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 TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (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 call DELETE$SEGMENT to delete the segment.

except$ptr — A POINTER to a WORD where the sequential condition 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, call A$SPECIAL with an open file connection, with spec$func equal to 0, and with ioparam$ptr pointing to a structure of the form:

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

To format a track on a disk, call A$SPECIAL with an open file connection, with spec$func equal to 0, and with ioparam$ptr pointing to a structure of the form:

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

In each of these structures, the fields are defined as follows:

track$number	The number of the track to be formatted. Acceptable values are 0 to one less than the number of tracks on the volume. Other values cause an E$SPACE exceptional condition.
interleave	The interleaving factor for the track. (That is, the number of physical sectors to advance when locating the next logical sector.) The supplied value, before being used, is evaluated mod the number of sectors per track.
track$offset	The number of physical sectors to advance when locating the first logical sector.
fill$char	The byte value with which each sector is to be filled.

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 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.

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 by the A$PHYSICAL$DETACH$DEVICE system call. 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.

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. It may be necessary to do a "hard" detach of the device.

3. Call A$PHYSICAL$ATTACH$DEVICE 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.

Obtaining Information About Winchesters and Certain Other Disks. This function applies only to physical files. If your device is a Winchester drive with an iSBC 215 disk controller or a drive with an iSBC 220 SMD controller, you can obtain specification information about it by calling A$SPECIAL with a token for a device connection, with spec$func set to 3, and with ioparm$ptr pointing to a structure of the form:

```
DECLARE disk$drive$data STRUCTURE(
  cylinders         WORD,
  fixed             BYTE,
  removable         BYTE,
  sectors           BYTE,
  sector$size       WORD,
  alternates        BYTE);
```

A$SPECIAL returns information to the fields of this structure, as follows:

cylinders	The total number of cylinders on the drive.
fixed	The number of heads on the fixed disk or Winchester disk.
removable	The number of heads on the removable disk cartridge.
sectors	The number of sectors in a track.
sector$size	The number of bytes in a sector.
alternates	The number of alternate cylinders on the drive.

Getting or Setting Attributes of a Terminal. These functions apply only to physical files. You can receive (get) or set the characteristics of a terminal that is being driven by the Terminal Device Driver by issuing a call to A$SPECIAL. In each case you supply a token for a connection to a terminal. To get the data, set spec$func equal to 4, and to set the data, set spec$func equal to 5. In each case, ioparm$ptr should point to a structure of the form:

```
DECLARE terminal$attributes STRUCTURE(
  num$words         WORD,
  num$used          WORD,
  connection$flags  WORD,
  terminal$flags    WORD,
  in$baud$rate      WORD,
  out$baud$rate     WORD,
  scroll$lines      WORD);
```

where:

num$words	The number of words, not including num$words and num$used, that are reserved for the remainder of the terminal$attributes data structure.

num$used	The number of fields, following the num$used field, that are actually being used for getting or setting terminal characteristics. If you are getting terminal information, the amount of data that is returned is governed by the num$used field. For example, if spec$func is 4 and num$used is 2, then connection$flags and terminal$flags will receive data but in$baud$rate, out$baud$rate, and scroll$lines will not. If you are setting terminal attributes, num$used specifies the number of <u>nonzero</u> words following the num$used field that are to be used for setting terminal attributes. For example, if num$used is 2, while connection$flags is 0 and terminal$flags is not 0, then the contents of terminal$flags will be used to set terminal attributes, but the contents of connection$flags <u>will be ignored</u>. In this way, you can set some parameters without affecting others.
connection$flags	This word applies only to this connection to the terminal. (All other parameters apply to the terminal itself and therefore to all connections to the terminal.) The flags in this word are encoded as follows. (Bit 0 is the low-order bit.)

Bits	Value and Meaning
0-1	Line editing control. (See Appendix F for a description of how to use a terminal. In particular, this appendix describes line editing.) 1 = No line editing. Input is transmitted to the requesting task exactly as entered at the terminal. Before being transmitted, data accumulates in a buffer until either a carriage return is entered or the requested number of characters has been entered. 2 = Line editing. Edited data accumulates in a buffer until either a carriage return is entered or the requested number of characters has been entered.

Bits	Value and Meaning
0-1	3 = No line editing. Input is transmitted to the requesting task exactly as entered at the terminal. Before being transmitted, data accumulates in a buffer until an input request is received. At that time, the contents of the buffer (or the number of characters requested, if the buffer contains more than that number) is transmitted to the requesting task. If any characters remain in the buffer, they are saved for the next input request.
2	Echo control. 0 = Echo. Characters entered into the terminal are "echoed" to the terminal's display screen. 1 = Do not echo.
3	Input parity control. Characters entered into the terminal have their parity bits (bit 7) set to 0 or not set, according to the value of the input parity control bit. 0 = Set parity bit to 0. 1 = Do not alter parity bit.
4	Output parity control. Characters being output to the terminal have their parity bits (bit 7) set to 0 or not set, according to the value of the output parity control bit. 0 = Set parity bit to 0. 1 = Do not alter parity bit.
5	Output control character control. This bit specifies whether output control characters are effective when entered at the terminal. The value of this bit applies only to output through this connection. Control characters are described in Appendix F of this manual. 0 = Accept output control characters in the input stream. 1 = Ignore output control characters in the input stream.

Bits	Value and Meaning
6-7	OSC control sequence control. These bits specify whether OSC control sequences should be acted upon when they appear in the input stream and, separately, when they appear in the output stream. These bits apply only to input or output through this connection. OSC control sequences are described in Appendix F of this manual. 0 = Act upon OSC sequences that appear in either the input or output stream. 1 = Act upon OSC sequences in the input stream only. 2 = Act upon OSC sequences in the output stream only. 3 = Do not act upon any OSC sequences.
8-15	Reserved bits. For future compatibility, set to 0.

terminal$flags   This word applies to the terminal and therefore to all connections to the terminal.

The flags in this word are encoded as follows. (Bit 0 is the low-order bit.)

Bits	Value and Meaning
0	Reserved bit. Set to 0.
1	Line protocol indicator. Full-duplex terminals support simultaneous and independent input and output. Half-duplex terminals support independent input and output, but not simultaneously. 0 = Full duplex. 1 = Half duplex.
2	Output medium. 0 = Video display terminal (VDT). 1 = Printed (Hard copy).

Bits	Value and Meaning
3	Modem indicator. 0 = Not used with a modem. 1 = Used with a modem.
4-5	Input parity control. The parity bit (bit 7) of each input byte can be used in a variety of ways. A byte has even parity if the sum of its bits is an even number. Otherwise, the byte has odd parity. 0 = Always set parity bit to 0. 1 = Never alter the parity bit. 2 = Even parity is expected on input. Use the parity bit to indicate the presence (1) or absence (0) of an error on input. That is, set the parity bit to 0 unless the received byte has odd parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.) 3 = Odd parity is expected in input. Use the parity bit to indicate the presence (1) or absence (0) of an error on input. That is, set the parity bit to 0 unless the received byte has even parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.)
6-8	Output parity control. The parity bit (bit 7) of each output byte can be used in a variety of ways. A byte has even parity if the sum of its bits is an even number. Otherwise, the byte has odd parity.

Bits	Value and Meaning
	0 = Always set parity bit to 0.
	1 = Always set parity bit to 1.
	2 = Set parity bit to give the byte even parity.
	3 = Set parity bit to give the byte odd parity.
	4 = Do not alter the parity bit.
9	Translation control. Translation refers to the ability to define certain control characters so that whenever these characters are entered at a terminal, certain actions, usually cursor movements, take place automatically. Translation is described in Appendix F of this manual.
	0 = Do not enable translation.
	1 = Enable translation.
10	Terminal axes sequence control. This specifies the order in which Cartesian-like coordinates of elements on a terminal's screen are to be listed or entered.
	0 = List or enter the horizontal coordinate first.
	1 = List or enter the vertical coordinate first.
11	Horizontal axis orientation control. This specifies whether the coordinates on the terminal's horizontal axis increase or decrease as you move from left to right across the screen.
	0 = Coordinates increase from left to right.
	1 = Coordinates decrease from left to right.
12	Vertical axis orientation control. This specifies whether the coordinates on the terminal's vertical axis increase or decrease as you move from top to bottom across the screen.

Bits	Value and Meaning
	0 = Coordinates increase from top to bottom.
	1 = Coordinates decrease from top to bottom.
13-15	Reserved bits. For future compatibility, set to 0.

NOTE

If bits 4-5 contain 2 or 3, and bits 6-8 also contain 2 or 3, then they must both contain the same value. That is, they must both reflect the same parity convention (even or odd).

in$baud$rate — This word contains an input baud rate indicator, which is encoded as follows:

0 = Not applicable.

1 = Perform an automatic baud rate search.

Other = Actual input baud rate, such as 2400.

out$baud$rate — This word contains an output baud rate indicator, which is encoded as follows:

0 = Not applicable

1 = Use the input baud rate for output.

Other = Actual output baud rate, such as 9600.

scroll$lines — An operator at a terminal can enter a control character (default is Control-W) when he/she is ready for data to appear on the terminal's display screen. The value contained in scroll$lines specifies the maximum number of lines that are to be sent to the terminal each time the control character is entered.

Designating Characters for Signalling from a Terminal Keyboard. You can use the A$SPECIAL system call to associate a keyboard character with a semaphore, so that whenever the character is entered into the terminal, the Basic I/O System automatically sends a unit to the semaphore. Up to 12 character-semaphore pairs can be so associated simultaneously, with each character being associated with a different semaphore, if desired.

To set up a character-semaphore pair, call A$SPECIAL with a device connection, with spec$func equal to 6, and with ioparm$ptr pointing to a structure of the form:

```
DECLARE signal$pair STRUCTURE(
  semaphore         WORD,
  character         BYTE);
```

where:

semaphore	A TOKEN for the semaphore that is to be associated with the character.
character	A hexadecimal value in the range 0 to 1FH. When the ASCII equivalent of this value is entered into the terminal, the Basic I/O System will send a unit to the associated semaphore.

To dissolve a semaphore-character relationship, make an identical call to A$SPECIAL, except that the semaphore field must contain 0.

CONDITION CODES

A$SPECIAL return condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection parameter is a connection produced by the Extended I/O System.

E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters or fields is not a token for an existing object: • The connection parameter • The resp$mbox parameter • The mailbox field in the notify structure. (Spec$func = 2.) • The object field in the notify structure. (Spec$func = 2.) • The semaphore field in the signal$pair structure. (Spec$func = 6.) 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$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$SPECIAL was not included when the system was configured.
E$PARAM	One or more of the following is true: • The spec$func parameter was 5, and one or more of the following is true: - Bits 0-1 of the connection$flags field was equal to 0. - Bits 6-8 of the terminal$flags field was greater than 4. • The spec$func parameter was 6, and the character field was greater than 1FH. • The spec$func parameter was greater than 6.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.

E$TYPE	One or more of the following parameters or fields was a token for an existing object of the wrong type: • The connection parameter. • The resp$mbox parameter. • The mailbox field of the notify structure. (Spec$func = 2.) • The semaphore field of the signal$pair structure. (Spec$func = 6.)

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	One or more of the following is true: • The function code is 'notify' and the connection is not a device connection. This applies only to named and physical files. • The connection is not open. This applies only to stream and physical files. • 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 Artificially Satisfying a Stream File I/O Request in the DESCRIPTION.)
E$FLUSHING	The connection to which this special function applies was closed before the function could be completed.
E$IFDR	The connection refers to a named file, but the function is not "notify".
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.
E$SPACE	The A$SPECIAL call attempted to format a physical file past the end of the device.

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 TOKEN for an open connection to the file being truncated.

OUTPUT PARAMETERS

resp$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition 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. File pointers for other connections to the file are not affected by the truncation operation.

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 must have update access to the file.

CONDITION CODES

A$TRUNCATE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection is a connection produced by the Extended I/O System.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$IFDR	This system call applies only to named files, but the connection parameter specified some other type of file.
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$TRUNCATE was not included when the system was configured.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.

E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The specified file is not open for writing or updating.
E$FACCESS	The connection does not have update access to the file.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.

A$UPDATE

A$UPDATE requests that the Basic I/O System write a partial sector that remains after the most recent A$WRITE call.

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

INPUT PARAMETERS

connection	A TOKEN for a connection to the file into which the partial sector is to be written.
resp$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.

OUTPUT PARAMETER

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

DESCRIPTION

The A$UPDATE call causes a partial sector of data to be written to the file specified by the connection parameter in the call. Such a call can be necessary because the I/O System, when performing an A$WRITE operation, writes only entire sectors. So if part of a sector remains to be written, the I/O System, unless requested to finish the writing operation (that is, to "update the file"), leaves the data for a partial sector in an output buffer. The next time A$WRITE is called on behalf of that file, the leftover data in the buffer is combined with the data in the new request, and the Basic I/O System again begins writing entire sectors. (Note that the A$UPDATE system call has no effect on buffers that the Extended I/O System manages.)

Three different events can cause the Basic I/O System to "update" a file. One, of course, is a call to A$UPDATE. The other two, called fixed updating and timeout updating, are triggered by the passing of (possibly different) amounts of time. You specify the time periods, and the devices to which they apply, when you configure the Basic I/O System.

Fixed updating occurs when an amount of time, which is specified for an entire application, passes. At that time, all devices to which updating applies are "updated". When configuring the Basic I/O System, you specify, for each I/O device, whether fixed updating applies to that device.

Timeout updating is just like fixed updating, except in two respects. First, the time period is defined separately for each device, rather than applying to the system as a whole. When configuring the Basic I/O System, you specify, for each I/O device, whether timeout updating applies to that device, and if it does, what the timeout period is to be for that device.

The second difference between timeout updating and fixed updating is that, in timeout updating, the timeout period commences at the beginning of each I/O operation, whereas fixed updating is independent of I/O activity.

CONDITION CODES

A$UPDATE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a <u>sequential</u> condition code. A code returned as a result of asynchronous processing is a <u>concurrent</u> condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$EXIST	Two conditions can cause this exception code to be returned: 1. One or more of the following parameters is not a token for an existing object: • The connection parameter • The resp$mbox parameter 2. The connection is being deleted.
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.

E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$NOT$CONFIGURED	A$UPDATE was not included when the system was configured.
E$SUPPORT	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$IO	An I/O error occurred, which might or might not have prevented the operation from being completed.

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 TOKEN for the open connection through which the write operation is to take place.
buff$ptr	A POINTER to the buffer that contains the data to be written.
count	A WORD giving the number of bytes to be written.

OUTPUT PARAMETERS

resp$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C). A value of zero means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it. If all the other connections to a stream file are requesting write operations, an actual value of zero and a status value of E$FLUSHING are returned in the I/O result segment.
except$ptr	A POINTER to a WORD where the sequential condition 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 location of the connection's 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.

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 commence writing. If a task does this, the Basic I/O System will extended the file to accommodate the writing operation. However, the data located between the old end of file and the beginning of the writing operation is undefined.

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 it must have append or update access to the file.

CONDITION CODES

A$WRITE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in Chapter 7 of this manual.

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

Sequential Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection is a connection produced by the Extended I/O System.
E$EXIST	Two conditions can cause this exception code to be returned:

1. One or more of the following parameters is not a token for an existing object:
   - The connection parameter
   - The resp$mbox 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.
E$NOT$CONFIGURED	A$WRITE was not included when the system was configured.
E$SUPPORT	The connection parameter specified is not valid in this system call because the connection was not created by this job.
E$TYPE	One or more of the following is true: • The connection parameter contained a token for an object that is not a connection. • The resp$mbox parameter contained a token for an object that is not a mailbox.

Concurrent Condition Codes

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

E$OK	No exceptional conditions.
E$CONTEXT	The connection is not open for writing or updating.
E$FACCESS	The connection does not have update or append access to the file.
E$FLUSHING	One or more of the following is true: • The connection was closed before the write operation could be performed. • 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, which might or might not have prevented the operation from being completed.

E$SPACE	One or more of the following is true: ● The volume has no more space. ● The operation attempted to write beyond the end of the device. This applies only to physical files.
E$SUPPORT	The write operation, if carried out, would extend the file, but as the Basic I/O System is configured, extending a file is not allowed.

CREATE$USER

The CREATE$USER system call creates a user object.

CAUTION

This system call overrides the protection mechanism provided by the Basic I/O System. It should be used only by system programmers in charge of security management.

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

INPUT PARAMETER

ids$ptr — A POINTER to a structure of the following form:

```
DECLARE  ids STRUCTURE(
   length      WORD,
   count       WORD,
   id(*)       WORD);
```

where:

length	Number of elements in the ID array.
count	Number of IDs (from the ID array) that are to be included in the user object. This number must be less than or equal to length, but greater than or equal to one.
id	Array of IDs, each of which is included in the user object. The first ID is to be used as the owner ID for any file created with reference to this user object.

OUTPUT PARAMETERS

user — A TOKEN where a token for the new user object will be returned.

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

DESCRIPTION

The CREATE$USER system call creates a user object. It accepts a list of IDs and returns a token for the new object.

If the number of ID slots, as specified by the length field, is greater than the number of IDs, as specified by the count field, the effect is as if length had been reduced to equal count.

CONDITION CODES

E$OK	No exceptional conditions.
E$LIMIT	The call cannot be processed without exceeding the maximum number (specified when the job was created) of objects allowed for this job.
E$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E$PARAM	The count field in the ids structure either is zero or is greater than the length field.

DELETE$USER

The DELETE$USER system call deletes a user object.

CAUTION

This system call overrides the protection mechanism provided by the Basic I/O System. It should be used only by system programmers in charge of security management.

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

INPUT PARAMETER

user	A TOKEN for the user object to be deleted.

OUTPUT PARAMETER

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

DESCRIPTION

The DELETE$USER system call deletes a user object. Deleting a user object has no effect on connections created with the user object.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The user parameter is not a token for an existing object.
E$LIMIT	The call cannot be processed without exceeding the number (255 decimal) of I/O operations which can be outstanding at one time for the user object specified in the call.

E$NOT$CONFIGURED	DELETE$USER was not included when the system was configured.
E$TYPE	The user parameter refers to an existing object of the wrong type.

GET$DEFAULT$PREFIX

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

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

INPUT PARAMETER

job	A TOKEN for the job whose default prefix is sought. A zero specifies the calling task's job.

OUTPUT PARAMETERS

connection	A TOKEN that receives a token for the connection object that is the default prefix for the designated job.
except$ptr	A POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CONDITION CODES

E$OK — No exceptional conditions.

E$NO$PREFIX — You specified a default prefix (prefix argument equals zero), but no default prefix can be found because of one of the following reasons:

- When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix.
- The job's directory can have entries but a default prefix is not cataloged there.

	• The prefix that is cataloged is not of 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.) • The job parameter contains a token for an object that is not a job.
E$NOT$CONFIGURED	GET$DEFAULT$PREFIX was not included when system was configured.

GET$DEFAULT$USER

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

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

INPUT PARAMETER

job	A TOKEN for the job whose default user object is sought. A zero specifies the calling task's job.

OUTPUT PARAMETERS

user$id	A TOKEN for the user object that is the default user for the designated job.
except$ptr	A POINTER to a WORD where the condition code will be returned.

DESCRIPTION

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

CONDITION CODES

E$OK	No exceptional conditions.
E$NO$USER	No default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but a default user is not cataloged there.

	• The object which is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
	• The job parameter contains a token for an object that is not a job.
E$NOT$CONFIGURED	GET$DEFAULT$USER was not included when the system was configured.

GET$TIME

The GET$TIME system call returns the system's date/time value.

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

OUTPUT PARAMETERS

date$time	A DWORD containing a date/time value expressed as the number of seconds since a fixed, user-determined point in time.
except$ptr	A POINTER to a WORD where the condition 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 as the number of seconds since some fixed, user-determined point in time. Any time in the past can be used as the "beginning of time".

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CONFIGURED	GET$TIME was not included when the system was configured.

INSPECT$USER

The INSPECT$USER system call returns a list of the IDs contained in a user object.

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

INPUT PARAMETER

user	A TOKEN for the user object being inspected.

OUTPUT PARAMETERS

ids$ptr — A POINTER to a structure of the following form:

```
DECLARE  ids  STRUCTURE(
   length        WORD,
   count         WORD,
   id(*)         WORD);
```

where:

length	Upper limit on the number of IDs that are to be returned.
count	Actual number of IDs that are being returned.
id(*)	The IDs being returned.

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

DESCRIPTION

The INSPECT$USER system accepts a token for a user object and returns a list of the IDs in the user object.

The calling task must supply the length value in the data structure pointed to by the ids$ptr parameter. The I/O System fills in the remaining fields in that structure.

If the length value is smaller than the actual number of IDs in the user object, only the specified number of IDs will be returned.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	The user parameter is not a token for an existing object.
E$NOT$CONFIGURED	INSPECT$USER was not included when the system was configured.
E$PARAM	The length field contains a value of zero.
E$TYPE	The job or user parameter refers to an object of the wrong type.

SET$DEFAULT$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 TOKEN for the job whose default prefix is to be set. A zero specifies the current job.
prefix	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 $.

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	When this job was created, a size of zero was specified for the object directory, so a default prefix cannot be cataloged
E$EXIST	One or more of the following parameters is not a token for an existing object: • The job parameter • The prefix parameter

E$LIMIT	The prefix parameter cannot be cataloged because the job object directory is full.
E$NOT$CONFIGURED	SET$DEFAULT$PREFIX was not included when the system was configured.
E$TYPE	One or more of the following is true: • The job parameter is a token for an object that is not a job. • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.)

SET$DEFAULT$USER

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

CAUTION

This system call overrides the protection mechanism provided by the Basic I/O System. It should be used only by system programmers in charge of security management.

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

INPUT PARAMETERS

job	A TOKEN for the job whose default user object is to be set. A zero designates the calling task's job.
user	A TOKEN for the user object that is to become the default user.

OUTPUT PARAMETER

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

DESCRIPTION

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

CONDITION CODES

E$OK	No exceptional conditions.
E$CONTEXT	When this job was created, a size of zero was specified for the object directory, so a default prefix cannot be cataloged.

E$EXIST	One or more of the following parameters is not a token for an existing object: • The job parameter • The user parameter
E$LIMIT	The user object cannot be cataloged because the job object directory is full.
E$NOT$CONFIGURED	SET$DEFAULT$USER was not included when the system was configured.
E$TYPE	The job or user argument refers to an object of the wrong type.

SET$TIME

CAUTION

This system call overrides the timing mechanism provided by the Basic I/O System. It should be used only by system programmers setting the initial system time.

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

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

INPUT PARAMETER

date$time	A DWORD containing a date/time value expressed as the number of seconds since a fixed, user-determined point in time.

OUTPUT PARAMETER

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

DESCRIPTION

The SET$TIME system call sets the date/time value for the I/O system. The I/O System maintains the date/time value a double word containing the number of seconds since a fixed point in time. Any time in the past can be used as the "beginning of time", but we recommend that you use 12:00 am (midnight), January 1, 1978.

CONDITION CODES

E$OK	No exceptional conditions.
E$NOT$CONFIGURED	SET$TIME was not included when the system was configured.

WAIT$IO

WAIT$IO can be called following a call to A$READ, A$WRITE, or A$SEEK. When called, it returns to the calling task the concurrent condition code for the prior call. If applicable, WAIT$IO also returns the number of bytes read or written.

```
actual = RQ$WAIT$IO(connection, resp$mbox, time$limit, except$ptr);
```

INPUT PARAMETERS

connection	A TOKEN for the connection that was specified as the connection in the prior asynchronous system call. (See DESCRIPTION.)
resp$mbox	A TOKEN for the mailbox that was specified as the response mailbox for the prior asynchronous system call. (See DESCRIPTION.)
time$limit	A WORD specifying the number of Nucleus system clock units that the task calling WAIT$IO is willing to wait for the I/O result segment to arrive at the response mailbox. A value of 0 means that the task is not willing to wait at all, and a value of 0FFFFH means that the task will wait indefinitely.

OUTPUT PARAMETERS

actual	A WORD to which the Basic I/O System returns the number of bytes read or written in the prior asynchronous system call. This value is undefined if the prior call was to A$SEEK. (See DESCRIPTION.)
except$ptr	A POINTER to a WORD where either the (concurrent) condition code for the prior asynchronous system call or the (sequential) condition code for the WAIT$IO system call is to be returned. (See DESCRIPTION.)

DESCRIPTION

There are two ways in which a task calling A$READ, A$WRITE, or A$SEEK can receive the result of the concurrent portion of the call from the designated response mailbox. One way is for the task to wait at the mailbox, receive an I/O result segment there, and extract the information from the segment. It is then incumbent upon the task to delete the segment, so that memory reserves are not needlessly depleted.

The other way for the task to receive this information is to call WAIT$IO. After the concurrent portion of the previous I/O call has been completed, the WAIT$IO system call returns the result of that call as follows:

- To the actual word, the number of bytes read or written, depending upon whether the previous call was to A$READ or A$WRITE, respectively. If the previous call was to A$SEEK, the value in the actual word is undefined.

- To the word pointed to by the except$ptr parameter, the concurrent condition code from the previous I/O call or the sequential condition code from the call to WAIT$IO. That is, if either if these condition codes is not E$OK, then that code is returned; if both of the condition codes are not E$OK, then the code that is returned is the code from the call to WAIT$IO. You should take note of the following:

  - There are four condition codes -- E$CONTEXT, E$LIMIT, E$MEM, and E$SUPPORT -- that can be returned by either the sequential or the concurrent portion of a system call. However, WAIT$IO does not return any of these codes, so if one of them is returned, it came from the previous I/O call.

  - If the concurrent portion of the previous I/O call caused an E$IO exceptional condition, this code is not returned. Instead (in this case only), WAIT$IO returns the value in the unit$status field of the I/O result segment for the previous I/O call. The possible values are the following and are described under CONDITION CODES:

Mnemonic	Value
E$IO$UNCLASS	50H
E$IO$SOFT	51H
E$IO$HARD	52H
E$IO$OPRINT	53H
E$IO$WRPROT	54H

The benefit of WAIT$IO is that, in applications that use it, tasks do not always have to deal directly with I/O result segments. In particular, those tasks do not have to delete I/O result segments. Because of this, the Basic I/O System, in applications using WAIT$IO, maintains a supply of I/O result segments that can be used repeatedly. This means that performance is enhanced because the Basic I/O System does not have to create a segment every time an I/O result segment is needed. This provides a significant advantage because A$READ, A$WRITE, and A$SEEK are typically the most commonly invoked Basic I/O System calls.

CONDITION CODES

E$OK	No exceptional conditions.
E$EXIST	One or more of the following is true: • In the call to WAIT$IO, the connection parameter or the resp$mbox parameter (or both) did not contain a token for an existing object. • The specified connection or response mailbox (or both) was deleted. • The token returned to the specified mailbox was for an object that had been deleted.
E$IO$HARD	A hard I/O error occurred. This means that a retry is probably useless.
E$IO$OPRINT	The device was off-line. Operator intervention is required.
E$IO$SOFT	A soft I/O error occurred. This means that the Basic I/O System tried several times to perform the asynchronous operation and failed, so another retry is probably useless.
E$IO$UNCLASS	An unknown type of I/O error occurred.
E$IO$WRPROT	The asynchronous operation was A$WRITE and the volume was write-protected.
E$NOT$CONFIGURED	WAIT$IO was not included when the system was configured.
E$TIME	One of the following is true: • The task calling WAIT$IO specified that it would not wait, and there was no I/O result segment at the specified mailbox. • The task calling WAIT$IO specified that it would wait for a specific period of time, but an I/O result segment was not sent to the response mailbox in that time period.

E$TYPE	One or more of the following is true: • The connection parameter was not a connection or was a device connection. • The resp$mbox parameter was not a mailbox. • The object received at the response mailbox was not a segment or was a segment that was not an I/O result segment.

***

CHAPTER 9. CONFIGURING THE BASIC I/O SYSTEM

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

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

The balance of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the iRMX 86 CONFIGURATION GUIDE.

BASIC I/O SYSTEM CALLS

You can select the system calls that your application requires. The advantage in being able to do this is that you can reduce the amount of Basic I/O System code needed to support your application. Moreover, if you choose to omit certain combinations of system calls, you can exclude entire file drivers, such as the stream file driver.

INTEL I/O DEVICES

You must specify which Intel I/O devices (controllers) are part of your hardware configuration. The devices that you can specify are listed in the iRMX 86 CONFIGURATION GUIDE.

For each device that you select, you must specify a name, physical characteristics, and desired operating modes.

BUFFERS

For each device, you must specify the number of buffers that the Basic I/O System is to manage during I/O operations on that device.

TIMING FACILITIES

You must specify whether you want your system to include the timing facilities related to the SET$TIME and GET$TIME system calls.

SERVICE TASK PRIORITIES

You must specify the priorities of the Basic I/O System tasks that attach devices and delete connections.

CREATION A FILE WITH AN EXISTING PATHNAME

Occasionally, a task will call A$CREATE$FILE, specifying a pathname that is identical to the pathname of a file that already exists. The Basic I/O System provides a configuration parameter (called NO_CREATE_FILE) that enables you to specify what should happen in this case.

If NO_CREATE_FILE is selected, the call to A$CREATE$FILE will return the exception code E$FEXIST, regardless of the value of the must$create parameter in the call.

If NO_CREATE_FILE is not selected, then what happens depends upon the value of the must$create parameter in the call to A$CREATE$FILE. If must$create is true (OFFH), then the Basic I/O System returns the E$FEXIST exception code. If must$create is false (0), then the existing file is truncated or expanded, according to the size parameter in the call to A$CREATE$FILE.

SYSTEM MANAGER ID

You must specify whether you want a system manager (user).

BASIC I/O SYSTEM IN ROM OR RAM

You must specify whether the Basic I/O System is to be in ROM or in RAM. If you specify RAM, as you will during the initial testing phase of your development cycle, you will have to arrange to load the Basic I/O System into memory.

FACTORS THAT AFFECT BASIC I/O SYSTEM PERFORMANCE

The purpose of this section is to make you aware of the factors that have the greatest impact on the performance (speed) of the Basic I/O System. Note that you determine some of these factors during software configuration, but you determine other factors at other times. The factors are as follows:

- Device granularity, which is the smallest number of bytes that can be read from or written to a device in a single I/O operation. If this value is selectable, you determine it either by jumpering hardware or by means of software, depending upon the device.

- Volume granularity, which is the smallest number of contiguous bytes that can be allocated from a volume in a single allocation. This value can vary from volume to volume and must be a multiple of the device granularity. You specify it when formatting the volume with the FORMAT command of the Human Interface or with the Files Utility.

- File granularity, which is the smallest number of bytes that can be allocated to a file in a single allocation. This value can vary from file to file and must be a multiple of the volume granularity. You specify each file's granularity when creating the file with the A$CREATE$FILE system call.

- The number of buffers for each device-unit. You specify this value when configuring the Basic I/O System.

- The number of bytes to be read or written. You specify this value in calls to A$READ and A$WRITE.

- The priority of tasks that the Basic I/O System supplies for the purpose of overseeing I/O operations. There is one such task for each device-unit; you specify the priority of each of these tasks when configuring the Basic I/O System.

- The amount of time between updates performed by the fixed update and timeout update features. You specify these time intervals when configuring the Basic I/O System. These two kinds of updating are explained in Chapter 8, in the description of the A$UPDATE system call.

For best results with these factors, you should begin by using your best judgment. Then, using the resulting performance figures as a base, you can experiment by changing a few (perhaps only one) factors at a time.

Obtaining the optimum combination of these factors is vital to the performance of any application of which I/O operations are a major part. Consequently, you would be wise to do some experimenting.

***

APPENDIX A. iRMX™ 86 DATA TYPES

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

BYTE	An unsigned, 8-bit, binary number.
WORD	An unsigned, 16-bit, binary number.
DWORD	An unsigned, 32-bit, binary number.
INTEGER	A signed, 16-bit, binary number that is stored in two's complement form.
OFFSET	An unsigned, 16-bit binary number whose value represents the distance from the base of an 8086 segment.
SELECTOR	An unsigned, 16-bit binary number whose value represents the base of an 8086 segment.
POINTER	Two adjacent WORDs containing the base of an 8086 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.
TOKEN	An unsigned, 16-bit, binary number that you must declare to be literally a WORD or a SELECTOR.

***

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$SEGMENT	006H
User	T$NUM$USER	100H
Connection	T$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$UPDATE
A$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 successfully 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 information about the remaining fields of the I/O result segment, refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O SYSTEMS.

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.

I/O RESULT SEGMENT

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.

SEQUENTIAL (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$STATE	7	7H
E$NOT$CONFIGURED	8	8H
E$SUPPORT	35	23H
E$DEV$OFFLINE	46	2EH
E$IFDR	47	2FH

SEQUENTIAL (PROGRAMMER ERROR) EXCEPTION CODES

CODE	DECIMAL	HEXADECIMAL
E$ZERO$DIVIDE	32768	8000H
E$OVERFLOW	32769	8001H
E$TYPE	32770	8002H
E$PARAM	32772	8004H
E$BAD$CALL	32773	8005H
E$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$ENTRY	36	24H
E$DIR$END	37	25H
E$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
E$ILL$VOL	45	2DH
E$IO$UNCLASS	80	50H
E$IO$SOFT	81	51H
E$IO$HARD	82	52H
E$IO$OPRINT	83	53H
E$IO$WRPROT	84	54H

***

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

You can assign a logical name to any device with Extended I/O System call LOGICAL$ATTACH$DEVICE. This creates a logical device object, (T$LOG$DEV) 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 E$DEV$OFF$LINE. 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 can return E$DEV$OFF$LINE. You can correct this situation by invoking at least one Extended I/O System call that references the logical device by its logical name (such as :F0:).

For further information, refer to the descriptions of LOGICAL$ATTACH$DEVICE, in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL, and A$PHYSICAL$ATTACH$DEVICE, in this manual.

***

APPENDIX F. USING THE iRMX™ 86 TERMINAL SUPPORT CODE

The iRMX 86 Operating System provides two software packages that you can use to interface an operator's terminal with an iRMX 86-based application system. One such package is the Terminal Handler, and it is described in the iRMX 86 TERMINAL HANDLER REFERENCE MANUAL. The other package is the iRMX 86 Terminal Support Code, and it is the subject of this appendix.

In addition to presenting different interfaces to terminal operators, there is one major difference between these two packages. The Terminal Handler is a complete package, providing all of the required software links between the terminal and the tasks that interact with the terminal. In contrast, the Terminal Support Code, although far more powerful, is not complete. Before you can use the Terminal Support Code in your operating system, you must provide a terminal device driver, which is the software link between the Terminal Support Code and the terminal. Intel supplies some of these drivers with the iRMX 86 Operating System.

The remainder of this appendix describes using and controlling a terminal that interfaces with the Terminal Support Code. Refer to Appendix G of this manual for information about the Intel-supplied terminal drivers. For information about writing a terminal device driver, consult the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 AND iRMX 88 I/O SYSTEMS.

CONTROLLING INPUT AND OUTPUT FROM A TERMINAL

By entering control characters, such as Control-P or Control-X, you can influence the manner in which data flows between the Basic I/O System portion of an iRMX 86 application system and a terminal that is connected to that system. On input, these characters perform line editing functions. On output, they stop the flow of data, resume the flow, or allow data to flow in bursts of one or more lines.

LINE EDITING AT A TERMINAL

Three buffers are involved when data is entered at a terminal. The first is a type-ahead buffer, where the Terminal Support Code places the data until a task calls A$READ to request input from the terminal. When the input request arrives, the Terminal Support Code transfers the data to a line buffer, where it edits the line (unless line editing is disabled) according to control characters that are intermixed with the data. If additional data is entered at the terminal, it goes directly into the line buffer for editing. When a line terminator is entered, the Terminal Support Code transfers the edited data to the third buffer, where the requesting task has access to it.

The Terminal Support Code automatically supplies the first two buffers, and the task supplies the third buffer in its call to A$READ.

In what follows, the term "current line" refers to the set of characters (possibly with editing having been performed on them) that have been entered after the most-recently-entered carriage return or line feed. It is not possible to say where the current line is, with respect to the three buffers, because this varies according to changing circumstances.

The control characters that the Terminal Support Code uses to edit data in the line buffer are described in the next few paragraphs. Each control character described here is the default, and each can be replaced with a different control character by means of a process that is explained later in this appendix.

Carriage Return Line Feed	Terminates the current line. Entering either of these causes both to be placed into the current line and also to be displayed at the terminal. After displaying the CR/LF sequence, the Terminal Support Code moves the current line (or the number of characters specified in the input request, if the request is for fewer characters than are in the current line) to the buffer specified in the input request. If characters remain in the line buffer, they will be used to fully satisfy the next request for input from the terminal.
Rubout	Rubs out the last data character in the current line. That is, the rubout character and the data character immediately preceding the rubout character in the current line are both removed from the current line. If the terminal has a display screen, the deleted character disappears from the display. If the terminal output is hard copy, the deleted character is displayed a second time, surrounded by the "#" character; for example, the sequence "CAT(rubout)(rubout)(rubout)" would appear as CAT#TAC# and would enter and remove the letters C, A, and T from the current line.
Control-P	Causes the next character entered to be treated as data, even if that character is normally a control character. The Control-P character, which is operative only when line editing is enabled, is not placed into the current line, and neither the Control-P nor the next character entered is displayed at the terminal.

Control-R	Causes the current line to be displayed with editing already performed. This enables the terminal operator to see the effects of the editing characters entered since the most recent line terminator. If the current line is empty, the previous line is displayed. Moreover, if an operator enters Control-R several times successively, the Terminal Support Code displays previous lines until it can't find any more lines; then it repeatedly displays the last line found until no more Control-R's are entered.
Control-U	Immediately empties the type-ahead buffers that the Terminal Support Code manages.
Control-X	Deletes the current line. This control character discards all characters entered since the most recent line terminator and causes "#" to be displayed.
Control-Z	Terminates the current line. Control-Z differs from Carriage Return and Line Feed in that Control-Z does not become part of the current line that it terminates. Consequently, entering Control-Z immediately after entering another line terminator causes the I/O result segment for the next input request to be returned with the value 0 in its ACTUAL field.

CONTROLLING OUTPUT TO A TERMINAL

When sending output to a terminal, the Terminal Support Code always operates in one of four modes. The current output mode can be switched dynamically to any of the other output modes. The output modes and their characteristics are as follows:

Normal	The Terminal Support Code accepts output from tasks and immediately passes the output to the terminal for display.
Stopped	The Terminal Support Code accepts output from tasks, but it queues the output rather than immediately passing it to the terminal.
Scrolling	The Terminal Support Code accepts output from tasks, and it queues the output as in the stopped mode. However, rather than completely preventing output from reaching the terminal, it sends a predetermined number (called the scrolling count) of lines to the terminal whenever an operator enters an appropriate control character at the terminal.

Discarding	The Terminal Support Code discards all output for the terminal, rather than queueing it or passing it to the terminal.

The following control characters, when entered at the terminal, change the output mode for the terminal. In addition, these control characters will be acted upon when they appear in the output stream, provided that the connection's connection$flags (see the description of A$SPECIAL in Chapter 8 of this manual) word so indicates.

As in the case of the input control characters, each control character described here is the default, and each can be replaced with a different control character by means of a process that is explained later in this appendix.

Control-O	Places output into or out of discarding mode. If the output is not in discarding mode, Control-O places output into discarding mode. On the other hand, if output is in discarding mode, Control-O places output into the mode it was in prior to entering discarding mode.
Control-Q	Places output into normal mode. However, if the last output control character was Control-S, the output mode returns to what it was before entering stopped mode. Note that this implies the following: • The Control-S, Control-Q sequence always returns the output mode to what it was before the sequence was begun. • The Control-Q, Control-Q sequence always places output into normal mode.
Control-S	Places output into stopped mode. However, if output was in the discarding mode, Control-S leaves it in discarding mode, but a subsequent Control-O will place it in stopped mode.
Control-T	Allows one output line to be sent to the terminal.
Control-W	Allows N lines to be sent to the terminal, where N is the current scrolling count.

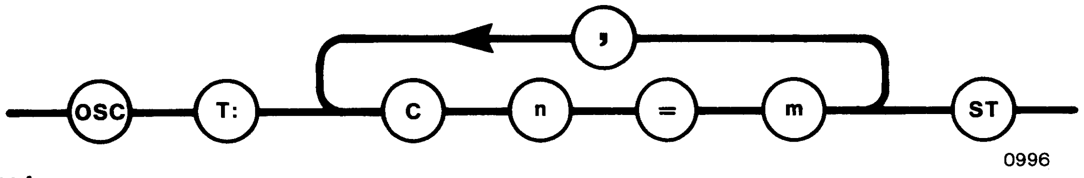
MODIFYING INPUT AND OUTPUT CONTROL CHARACTER ASSIGNMENTS

As indicated in the previous sections, control character assignments can be altered dynamically. The mechanism for this kind of change is a Software Control String, as defined in the American National Standards Institute publication ANSI X3.64 (1979). In this appendix, we are concerned with two kinds of Software Control Strings.

The first kind of Software Control String is used in communication from a program or terminal to an operating system. The opening delimiter for such a string is Escape Right Bracket (Esc]) and the closing delimiter is Escape Backslash (Esc \). This appendix calls this kind of string an Operating System Command sequence, or OSC sequence. Furthermore, the appendix uses the abbreviation OSC to stand for the opening delimiter and ST to stand for the closing delimiter.

The other kind of Software Control String is used in communication from an operating system to an application program. The opening delimiter for this kind of string is Escape Underline (Esc _) and the closing delimiter, as in OSC sequences, is Escape Backslash (Esc \). This appendix calls this kind of string an Application Program Command sequence, or APC sequence. The appendix uses the abbreviation APC to stand for the opening delimiter of an APC sequence.

Under conditions that are described later in this section, input and output control character assignments are altered by means of OSC sequences of the form



where:

T	An abbreviation for any word, such as Terminal, that starts with that letter.
C	An abbreviation for any word, such as Control, that starts with that letter.
n	The decimal representation of the ASCII code for the desired control character.
m	A function code from Table F-1.

The following sequence cancels the default assignment of Rubout (DEL) as the deletion character and assigns Backspace (BS) in its place:

```
OSC TERM: C127=0, C8=11 ST
```

MODES OF TERMINAL OPERATION

A terminal that is being supported by the Terminal Support Code is governed by numerous modes of operation. Some of these modes apply directly to the terminal, and are independent of the connection that a task uses to communicate with the terminal. The remaining modes depend entirely upon the connection being used.

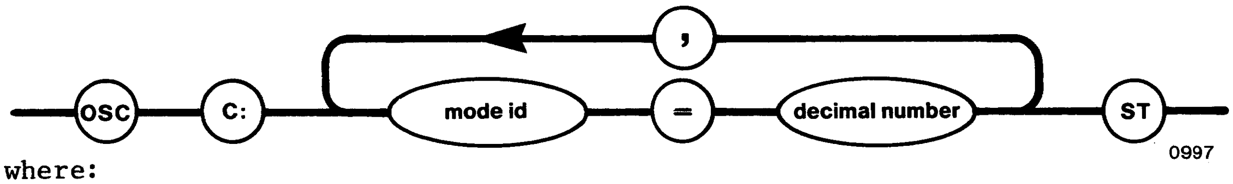
Table F-1. Menu of Control Character Functions

m	Abbreviated Functional Description	Default Assignment
0	None	None
1	Stop output	Control-S
2	Start output	Control-Q
3	Discard output	Control-O
4	Scroll N lines	Control-W
5	Scroll 1 line	Control-T
6	Empty type-ahead buffer	Control-U
7	Escape	Escape
8	CR/LF line terminator	Control-J, Control-M
9	Line terminator without CR/LF	Control-Z
10	Accept next character literally	Control-P
11	Delete character (Rubout)	Rubout
12	Cancel line	Control-X
13	Reprint line with editing	Control-R
14	Line terminator without CR/LF	None

MODES THAT A TERMINAL INHERITS FROM A CONNECTION

This appendix discusses the modes that a terminal inherits from a connection first, because they are relatively few in number and are easy to understand. Each of these modes is directly related to one or more bits in the connection$flags word for the connection. The full definition of each portion of the connection$flags word is provided in Chapter 8 of this manual, in the description of the A$SPECIAL system call. The names of the modes, the single-letter identification codes for the modes, the bits of the connection$flags word to which they correspond, and a brief description of their functions are given in Table F-2.

Assuming that the OSC control mode is set appropriately, the modes that a terminal inherits from a connection can be altered. The syntax of an OSC sequence that will change one or more of these modes is as follows:



where:

C	An abbreviation for any word, such as Connection, that begins with that letter.
mode id	An ID letter from the list of modes given in Table F-2.
decimal number	The value to which you want to change the mode.

For these decimal number values, refer to the description of A$SPECIAL in Chapter 8.

Table F-2. Inherited Terminal Modes

Mode Name	ID	Bit(s)	Description
Line editing	T	0-1	Indicates whether line editing is to be enabled. If not, indicates whether input requests should wait for a line terminator to be entered.
Echo	E	2	Indicates whether characters entered at the terminal are to be echoed to the display.
Input parity setting	R	3	Indicates whether characters entered at the terminal are to have their parity bits set.
Output parity setting	W	4	Indicates whether characters destined for the terminal are to have their parity bits set.
Output control	O	5	Indicates whether output control characters are to be recognized and acted upon when they are entered at the terminal.
OSC control	C	6-7	Indicates, for input and output (separately), whether OSC control sequences should be recognized and acted upon.

NOTE

It is possible to use two or more connections concurrently for obtaining input from a terminal. When this is the case, an operator at the terminal cannot always be certain as to which connection is being used to read the characters the operator is entering. In this case the operator cannot tell which connection's modes are being altered when the operator enters an OSC sequence as described in this section. To avoid this problem, never use multiple connections concurrently for input when you are planning to use OSC sequences to alter connection modes.

MODES THAT BELONG TO A TERMINAL (PART 1)

A terminal has several more modes of its own than modes that it has inherited from a connection. This section discusses some of these modes. Then, after covering the subjects of translation and simulation, a later section returns to the subject of modes that a terminal owns and completes the coverage of it.

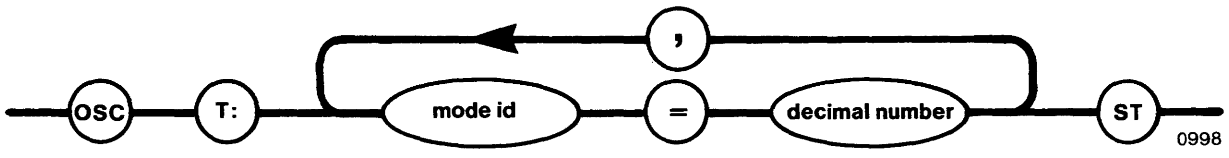
The structure of this section is similar to that of the previous section, but the focus is now on the terminal rather than on a connection to the terminal. Consequently, this section is based on the fact that each of the terminal modes is directly related to one or more bits in the terminal$flags word for the terminal. The full definition of each portion of the terminal$flags word is provided in Chapter 8 of this manual, in the description of the A$SPECIAL system call. The names of the modes, the single-letter identification codes for the modes, the bits of the terminal$flags word to which they correspond, and a brief description of their functions are given in Table F-3.

Table F-3. Non-Inherited Terminal Modes in Terminal$Flags (Part 1)

Mode Name	ID	Bit(s)	Description
Line protocol	L	1	Indicates whether the terminal is half-duplex or full-duplex.
Output medium	H	2	Indicates whether the terminal has a display screen or produces hard copy (printed) output.
Modem indicator	M	3	Indicates whether the terminal is connected to the hardware by a modem.
Input parity handling	R	4-5	Indicates how parity is to be interpreted and altered on input.
Output parity handling	W	6-8	Indicates how parity is to be interpreted and altered on output.

In addition to bits in the terminal$flags word for the terminal, three WORD parameters belong in this section. These parameters, also covered in the description of A$SPECIAL, are described in Table F-4.

Assuming that the OSC control mode is set appropriately, a terminal's modes can be altered. The syntax of an OSC sequence that changes one or more of the modes covered in this section is as follows:



where:

T	An abbreviation for any word, such as Terminal, that begins with that letter.
mode id	An ID letter from the list of modes given in Table F-3 or F-4.
decimal number	The value to which you want to change the mode. For these decimal number values, refer to the description of A$SPECIAL in Chapter 8.

Table F-4. Other Non-Inherited Terminal Modes (Part 1)

Mode Name	ID	Description
Input baud rate	I	The input baud rate that the terminal is configured to handle.
Output baud rate	O	The output baud rate that the terminal is configured to handle.
Scrolling number	S	The number of lines of output that are to be sent to the terminal's display whenever the scrolling control character (default is Control-W) is entered at the terminal.

TRANSLATION AND SIMULATION

The Terminal Support Code supports both translation and simulation by equating short commands (called terminal character sequences) with longer commands (called escape sequences.)

Translation

Translation occurs when a task calls A$WRITE to write an escape sequence through a connection to a terminal. The Terminal Support Code, instead of simply passing the escape sequence on to the terminal, intercepts the escape sequence and sends the equivalent terminal character sequence to the terminal in place of the escape sequence. The terminal can understand the terminal character sequence, and it responds as the task had intended it to respond.

An example of a terminal character sequence that can be used in translation is a code (such as Control-H) that tells the terminal to move the cursor backward by one position. After the necessary steps (described shortly) have been taken, if a task writes the appropriate escape sequence to the terminal, the Terminal Support Code intercepts it and replaces it with the Control-H that the terminal interprets as a signal to move its cursor backward by one position.

Translation makes application systems easily adaptable to many terminal brands and models. When one type of terminal is replaced with another type, only the association between an escape sequence and a terminal character sequence needs to be changed.

Simulation

Simulation occurs when a task calls A$WRITE to write an escape sequence that the terminal <u>does not</u> recognize. The Terminal Support Code intercepts the escape sequence and figures out what the task wants the terminal to do. Then the Terminal Support Code sends a series of one or more terminal character sequences that the terminal <u>does</u> recognize, producing the effect that the task wanted.

An example of simulation concerns tab stops. If a terminal does not support tab stops, the Terminal Support Code, if given the right information about the terminal, can simulate the tab stops, creating the impression that the terminal does indeed support tab stops as if it were a typewriter. All that the Terminal Support Code must do to accomplish this is to:

- Remember where the cursor is on the display.
- Remember where the tab stops are supposed to be.
- Be able to tell the terminal to move the cursor forward by one space.

Terminal Capabilities Required to Support Simulation

As was just shown, in order to simulate tab stops the Terminal Support Code needs to know how to tell a terminal to move the cursor forward by one space. As it happens, this comes close to answering the general question of what capabilities the terminal must have to support simulation. The abilities that the terminal must have are that it must be able to move its cursor as follows:

- One position to the right
- One position to the left
- One position upward
- One position downward
- To the upper-left corner of the display

Specifying the Desired Translation and Simulation Functions

As was shown, both translation and simulation are made possible by establishing a one-to-one correspondence between escape sequences and terminal character sequences. What has not yet been answered is the question of how this correspondence is defined.

The answer is that each pairing in this correspondence must be established individually by means of an OSC sequence, although an OSC sequence can establish multiple pairings. Depending upon the circumstances, the OSC sequence can be entered into the terminal or it can be issued from a task.

In order for an operator to establish a pairing by entering the OSC sequence into the terminal, the following conditions must exist:

- There must be a connection to the terminal, it must be open for reading, and there must be an A$READ request awaiting fulfillment at that connection.
- The OSC control bits for that connection must be set to permit the Terminal Support Code to recognize and act upon OSC sequences on input.
- The line-editing control bits for that connection must be set to permit line editing.

When these conditions exist, an operator may enter the OSC sequence (which will be described soon) into the terminal.

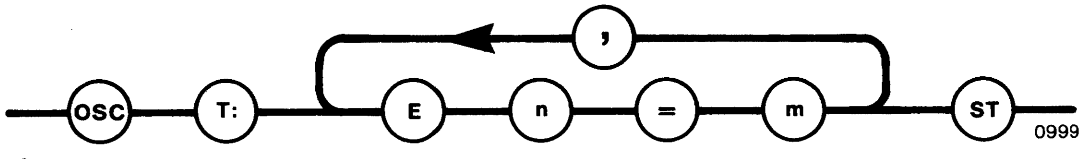
In order for a task to establish a pairing, the following conditions must exist:

- There must be a connection to the terminal, and it must be open for writing.
- The OSC control bits for that connection must be set to permit the Terminal Support Code to recognize and act upon OSC sequences on output.

When these conditions exist, a task may call A$WRITE to send the OSC sequence to the terminal.

Regardless of whether the OSC sequences came from a task or from the terminal, the Terminal Support Code intercepts the OSC sequence (thereby preventing it from going any further) and establishes the desired pairing.

The syntax of an OSC sequence that will establish one or more escape sequence-terminal character sequence pairings is as follows:



where:

T	An abbreviation for any word, such as Terminal, that begins with that letter.
E	An abbreviation for any word, such as Escape, that begins with that letter.
n	The number of an escape sequence listed in Table F-5.
m	The number of a terminal character sequence listed in Table F-6.

Table F-5 lists the escape sequences that can be paired with a terminal character sequence by means of OSC sequences. The following remarks apply to Table F-5:

- The code column contains codes used in the ANSI X3.64 document.
- The expression "99" represents any decimal number. Unless otherwise specified, the number can be omitted and the Terminal Supply Code supplies the default value 1.
- In some cases, multiple escape sequences can be combined into a single, compound escape sequence. These cases are identified in the table.
- The Terminal Support Code can simulate the escape sequences numbered 0, 1, 6 through 11, 13, 15, 16, 18 through 20, 22, and 23. The remaining escape sequences can only be translated.
- In almost all cases, tasks issue the escape sequences by calling A$WRITE. The exceptions concern escape sequences 7 and 18, and they are described in the table.

Table F-5. Escape Sequences

n	Code	Escape Sequence	Function
0	HTS	Esc H	Sets a horizontal tab at the current cursor position.
1	RIS	Esc c	Returns the terminal to its initial state. This consists of resetting the horizontal tab stops to four spaces apart, beginning with the first space, and returning the cursor to the upper-left corner of the display.
2	CUF	Esc [99 C	Moves the cursor forward the specified number of positions.
3	CUB	Esc [99 D	Moves the cursor backward the specified number of positions.

Table F-5. Escape Sequences (continued)

n	Code	Escape Sequence	Function
4	CUU	Esc [99 A	Moves the cursor upward the specified number of positions.
5	CUD	Esc [99 B	Moves the cursor downward the specified number of positions.
6	CUP	Esc [99 ; 99 H	Moves the cursor to the position specified by the decimal numbers. The first number specifies the horizontal coordinate position, and the second number specifies the vertical coordinate position. The horizontal coordinates are numbered from left to right, beginning with 1, and the vertical coordinates are numbered from top to bottom, also beginning with 1. If the parameters are omitted, this sequence moves the cursor to the upper-left corner of the display.
7	CPR	Esc [99 ; 99 R	Reports the coordinates of the current cursor position. The Terminal Support Code sends this sequence in response to sequence number 19, which asks for the cursor's coordinates. The first number specifies the horizontal coordinate position, and the second number specifies the vertical coordinate position. The horizontal coordinates are numbered from left to right, beginning with 1, and the vertical coordinates are numbered from top to bottom, also beginning with 1.
8	CBT	Esc [99 Z	Moves the cursor backward by the specified number of horizontal tab stops. For example, if the specified number is 2, the cursor moves backward to the second tab stop that it encounters.
9	CHA	Esc [99 G	Moves the cursor to the specified position in the line where the cursor is currently located.
10	CHT	Esc [99 I	Moves the cursor forward by the specified number of horizontal tab stops. For example, if the specified number is 2, the cursor moves forward to the second tab stop that it encounters.

Table F-5. Escape Sequences (continued)

n	Code	Escape Sequence	Function
11	CTC	Esc [0 W	Sets a horizontal tab stop at the current cursor position. This or any other CTC escape sequence can be combined with one or more CTC escape sequences to form a compound CTC escape sequence. An example of such a combined sequence is ESC [0;1 W, which sets both horizontal and vertical tab stops at the cursor position.
12	CTC	Esc [1 W	Sets a vertical tab stop at the current cursor position. See the description of escape sequence number 11.
13	CTC	Esc [2 W	Clears a horizontal tab stop if there is one at the current cursor position. See the description of escape sequence number 11.
14	CTC	Esc [3 W	Clears a vertical tab stop is there is one at the current cursor position. See the description of escape sequence number 11.
15	CTC	Esc [4 W	Clears all horizontal tab stops on the line where the cursor is located. See the description of escape sequence number 11.
16	CTC	Esc [5 W	Clears all horizontal and vertical tab stops. See the description of escape sequence number 11.
17	CTC	Esc [6 W	Clears all vertical tab stops. See the description of escape sequence number 11.
18	DA	Esc [99 c	Tasks send this sequence with the number 0 to request the ID number of the terminal to which the request is being sent. The Terminal Support Code intercepts the request and returns to the requesting task an identical sequence, except that the number (which is greater than 0) is the requested ID number.
19	DSR	Esc [6 n	Asks the Terminal Support Code to report the coordinates of the current cursor position. See sequence number 7 for a description of the response.

Table F-5. Escape Sequences (continued)

n	Code	Escape Sequence	Function
20	TBC	Esc [0 g	Clears a horizontal tab stop if there is one at the current cursor position. This or any other TBC escape sequence can be combined with one or more TBC escape sequences to form a compound TBC escape sequence. An example of such a combined sequence is ESC [0;1 g, which clears both horizontal and vertical tab stops from the current cursor position.
21	TBC	Esc [1 g	Clears a vertical tab stop if there is one at the current cursor position. See the description of escape sequence number 20.
22	TBC	Esc [2 g	Clears all horizontal tab stops on the line where the cursor is located. See the description of escape sequence number 20.
23	TBC	Esc [3 g	Clears all horizontal and vertical tab stops. See the description of escape sequence number 20.
24	TBC	Esc [4 g	Clears all vertical tab stops. See the description of escape sequence number 20.
25	DCH	Esc [99 P	Deletes the specified number of characters, beginning at the current cursor location.
26	DL	Esc [99 M	Deletes the specified number of lines, beginning at the line where the cursor is located.
27	ECH	Esc [99 X	Replace the specified number of characters with blanks, beginning at the current cursor location.
28	ED	Esc [0 J	Places blanks at all positions from the cursor to the end of the display. This or any other ED escape sequence can be combined with one or more ED escape sequences to form a compound ED escape sequence. An example of such a combined sequence is ESC [0;1 J, which clears the entire display.
29	ED	Esc [1 J	Places blanks at all positions from the beginning of the display to the cursor. See the description of escape sequence number 28.

Table F-5. Escape Sequences (continued)

n	Code	Escape Sequence	Function
30	ED	Esc [2 J	Fills the entire display with blanks. See the description of escape sequence number 28.
31	EL	Esc [0 K	Places blanks at all positions from the cursor to the end of the line This or any other EL escape sequence can be combined with one or more EL escape sequences to form a compound EL escape sequence. An example of such a combined sequence is ESC [0;1 K, which places blanks throughout the line containing the current cursor position.
32	EL	Esc [1 K	Places blanks at all positions from the beginning of the line containing the cursor to the cursor itself. See the description of escape sequence number 31.
33	EL	Esc [2 K	Places blanks at all positions in the line containing the cursor. See the description of escape sequence number 31.
34	ICH	Esc [99 @	Inserts the specified number of blanks, beginning at the location of the cursor.
35	IL	Esc [99 L	Inserts the specified number of blank lines, beginning at the location of the cursor.
36	NP	Esc [99 U	Moves the display forward in a multiple-page file by the specified number of pages. If the specified number of pages is 0, the display moves to the next page.
37	PP	Esc [99 V	Moves the display backward in a multiple-page file by the specified number of pages. If the specified number of pages is 0, the display moves to the previous page.
38	SD	Esc [99 T	Moves the display downward (forward) by the specified number of lines. If the specified number of lines is 0, the display moves to the next line.
39	SU	Esc [99 S	Moves the display upward (backward) by the specified number of lines. If the specified number of lines is 0, the display moves to the previous line.
40		Reserved	

Table F-5. Escape Sequences (continued)

n	Coce	Escape Sequence	Function
41	RM	Esc [0 1	An error condition.
42	RM	Esc [1 1	See the comment following this table.
43	RM	Esc [2 1	Unlocks the terminal's keyboard, allowing all characters to be input when typed.*
44	RM	Esc [3 1	Causes control characters not to be displayed, but still causes those characters to have their normal effects.*
45	RM	Esc [4 1	Causes characters that are output to overwrite characters on the display.*
46	RM	Esc [5 1	See the comment following this table.
47	RM	Esc [6 1	See the comment following this table.
48	RM	Esc [7 1	See the comment following this table.
49	RM	Esc [8 1	Reserved.
50	RM	Esc [9 1	Reserved.
51	RM	Esc [10 1	See the comment following this table.
52	RM	Esc [11 1	See the comment following this table.
53	RM	Esc [12 1	Causes characters to be displayed on the terminal's display screen as they are entered.
54	RM	Esc [13 1	See the comment following this table.
55	RM	Esc [14 1	See the comment following this table.
56	RM	Esc [15 1	See the comment following this table.
57	RM	Esc [16 1	See the comment following this table.
58	RM	Esc [17 1	See the comment following this table.
59	RM	Esc [18 1	Causes horizontal tab stops to apply equally to all lines, rather than on a line-by-line basis.*

* This is the normal (default) setting for most terminals.

Table F-5. Escape Sequences (continued)

n	Code	Escape Sequence	Function
60	RM	Esc [19 l	Causes data on the terminal's display screen to be treated as a continuous stream, rather than as a collection of disjoint, independent pages.*
61	RM	Esc [20 l	Prevents the line feed character from automatically performing a carriage return when output to the terminal.*
62	SM	Esc [0 h	An error condition.
63	SM	Esc [1 h	See the comment following this table.
64	SM	Esc [2 h	Locks the terminal's keyboard, preventing characters from being input when they are typed.
65	SM	Esc [3 h	Enables the display of control characters for debugging purposes.
66	SM	Esc [4 h	Enables output characters to be inserted in the display, rather than always overwriting existing characters
67	SM	Esc [5 h	See the comment following this table.
68	SM	Esc [6 h	See the comment following this table.
69	SM	Esc [7 h	See the comment following this table.
70	SM	Esc [8 h	Reserved.
71	SM	Esc [9 h	Reserved.
72	SM	Esc [10 h	See the comment following this table.
73	SM	Esc [11 h	See the comment following this table.
74	SM	Esc [12 h	Prevents characters from being displayed on the terminal's screen as they are typed.
75	SM	Esc [13 h	See the comment following this table.
76	SM	Esc [14 h	See the comment following this table.
77	SM	Esc [15 h	See the comment following this table.

* This is the normal (default) setting for most terminals.

Table F-5. Escape Sequences (continued)

n	Code	Escape Sequence	Function
78	SM	Esc [16 h	See the comment following this table.
79	SM	Esc [17 h	See the comment following this table.
80	SM	Esc [18 h	Causes horizontal tab stops to apply only to the line on which they are entered.
81	SM	Esc [19 h	Causes data to be treated as a collection of disjoint, independent pages. In this kind of environment, a terminal operator typically accesses the various pages in a file by pressing keys such as "next page", "previous page", or "go to page".
82	SM	Esc [20 h	Causes the line feed character to automatically perform a carriage return when output to the terminal.

Comment: This mode (or sequence or function) is included for completeness, but a description is beyond the scope of this manual. For details, refer to the 1979 version of the ANSI X3.64 standard.

Table F-6 is a list of the terminal character sequences that can be paired with escape sequences by means of OSC sequences. Recall that the assignment portion of such an OSC sequence has the form En=m, where n is the number of an escape sequence and m is the number of a terminal character sequence. In fact, there are three exceptions to this generality, each of which is described in Table F-6.

Earlier in this appendix, there are two brief examples, one concerning translation and one concerning simulation. You now have the tools to better understand these examples.

In the first example, which illustrated translation, Control-H (m=8) was suggested as a terminal character sequence that would cause the terminal to move its cursor backward one position. From Table F-5, we see that the escape sequence for moving a cursor backward by one position is "Esc [D" (n=3). To define the relationship between m=8 and n=3 for the Terminal Support Code, the OSC sequence:

```
OSC T:E3=8 ST
```

either can be entered at the terminal or can be written (to the terminal) by a task, depending upon the circumstances described earlier. Once this relationship is defined, any time a task writes the escape sequence "Esc [D" to the terminal, the terminal's cursor moves backward one position. Moreover, if a task writes that escape sequence with a "repeat factor", as in "Esc [6 D", the cursor moves backward by the appropriate number (in this case 6) positions.

Table F-6. Terminal Character Sequences

m	Terminal Character Sequence or Special Instructions
0	Disable the translation of escape sequence n. That is, pass the escape sequence through to the terminal without modification.
1	01H (Control-A)
2	02H (Control-B)
•	
•	
•	
26	1AH (Control-Z)
27	1BH (Esc)
28	1CH (FS)
29	1DH (GS)
30	1EH (RS)
31	1FH (US)
32	Esc 00H
33	Esc 01H
•	
•	
•	
159	Esc 7FH
160-191	Reserved
192	Simulate the escape sequence
193	Discard the escape sequence. That is, do not translate or simulate it, and do not pass it on to the terminal.

The second example used cursor movements to simulate tab stops. Suppose the terminal accepts Control-G as the terminal character sequence that moves the cursor forward by one position. Suppose further that application tasks wish to use CHT (n=10) to advance the cursor to the next tab stop, and CTC (n=11) to set a tab stop. As you can verify by looking at Tables F-5 and F-6, sending the following OSC sequence defines all of these relationships for the Terminal Support Code:

```
OSC T:E2=7, E11=192, E10=192 ST
```

In addition, it is necessary to advise the Terminal Support Code of the location of the cursor. You can do this by resetting the terminal, that is, by sending the sequence Esc c to the terminal. Having done these things, you can set a horizontal tab stop by entering Esc [0 W at the terminal, and you can advance the cursor to the next tab stop by entering Esc [1 I. The Terminal Support Code keeps track of the locations of the horizontal tab stops as well as the position of the cursor.

MODES THAT BELONG TO A TERMINAL (PART 2)

As you might have guessed, before the Terminal Support Code can monitor or control the position of a terminal's cursor, it must be informed of the coordinate numbering conventions for that terminal. The Terminal Support Code has its own "model" of a terminal's coordinate numbering scheme. As was mentioned in Table F-5, this model is the following: The horizontal coordinates are numbered from left to right, beginning with 1, and the vertical coordinates are numbered from top to bottom, also beginning with 1. Although this seems a perfectly reasonable way to reference positions on a terminal's screen, it is not universally applicable. Consider the following example, which is not as contrived as it might seem.

Suppose that a terminal's horizontal positions (that is, its columns) are numbered, left to right, as 80, 81, 82, ..., 127, 16, 17, 18, ..., 31. And suppose that its vertical positions (its rows) are numbered, top to bottom, as 103, 102, 101, ..., 80. And, finally, suppose that when referencing a particular position on the terminal's screen, you must specify the vertical position first, followed by the horizontal position. Note that this numbering convention differs from the Terminal Support Code's numbering convention in the following ways:

- The numbering on each axis starts with 80, rather than starting with 1.
- The numbering of the horizontal axis, when it reaches 127, drops back to 16 before resuming its climb.
- The numbering of the vertical axis increases from bottom to top, rather than increasing from top to bottom.
- The coordinates of a given screen position are vertical coordinate first, then horizontal coordinate, rather than being horizontal first and vertical second.

In fact, the only resemblances between the Terminal Support Code's convention and this terminal's convention are:

- The numbering of both axes start with the same value and then increase by ones (except for the drop from 127 to 16.)
- The numbering of the horizontal axis increases from left to right.

As bizarre as this example seems, it still falls within the requirements of the Terminal Support Code. These requirements can be stated as follows:

> The numbering of both axes must start (at the left or right, and at the top or bottom, it doesn't matter) with the same positive value. From there, they must increase by ones until (or unless) they reach 127. After reaching 127, each must "fall back" to a lower positive value, whereafter they must again increase by ones. If they both reach 127, they must both fall back to the same value.

Having specified all of this, consider the question: How is it possible to describe a terminal's numbering convention for the Terminal Support Code? The answer is that each terminal has modes other than those discussed in the earlier section that describes modes that a terminal owns. Four of these modes are designated in bits 9, 10, 11, and 12 of the terminal$flags WORD. (The full definition of each portion of the terminal$flags word is provided in Chapter 8 of this manual, in the description of the A$SPECIAL system call.) The names of these modes, their single-letter identification codes, the bits of the terminal$flags word to which they correspond, and a brief description of their functions are given in Table F-7.

Table F-7. Non-Inherited Terminal Modes in Terminal$Flags (Part 2)

Mode Name	ID	Bit(s)	Description
Translation	T	9	Indicates whether the Terminal Support Code for this terminal will be called upon to perform translation between escape sequences and terminal character sequences.
Terminal axis sequence	F*	10	Indicates whether the horizontal or the vertical coordinate is to be called out first when referencing a position on the terminal's screen.
Horizontal axis orientation	F*	11	Indicates whether the numbering of coordinates on the horizontal axis increases from left to right or from right to left.
Vertical axis orientation	F*	12	Indicates whether the numbering of coordinates on the vertical axis increases from top to bottom or from bottom to top.

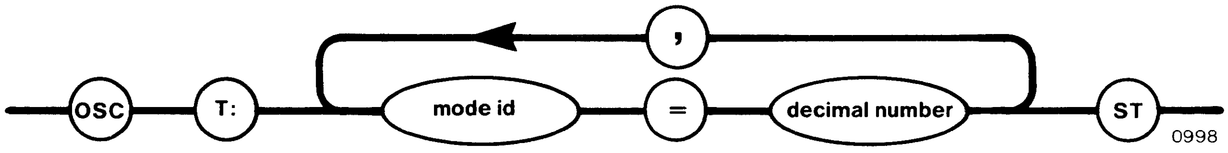
*This is the only case where one letter denotes more than one terminal mode.

In addition to bits in the terminal$flags word for the terminal, four BYTE parameters belong in this section. These parameters, which are <u>not</u> covered in the description of A$SPECIAL, are described in Table F-8.

Table F-8. Other Non-Inherited Terminal Modes (Part 2)

Mode Name	ID	Description
Cursor addressing offset	U	The value that is used to start the numbering sequence on both axes.
Overflow offset	V	The value to which the numbering of the axes must "fall back" after reaching 127.
Screen width	X	The number of character positions on each line of the terminal's screen.
Screen Height	Y	The number of lines on the terminal's screen.

Assuming that the OSC control mode is set appropriately, a terminal's modes can be altered. The syntax of an OSC sequence that will change one or more of the modes covered in this section is as follows:



where

T	An abbreviation for any word, such as Terminal, that begins with that letter.
mode id	An id letter from the list of modes given in Table F-7 or F-8.
decimal number	The value to which you want to change the mode. For information about the decimal number values that pertain to the terminal$flags word, refer to the description of A$SPECIAL in Chapter 8.

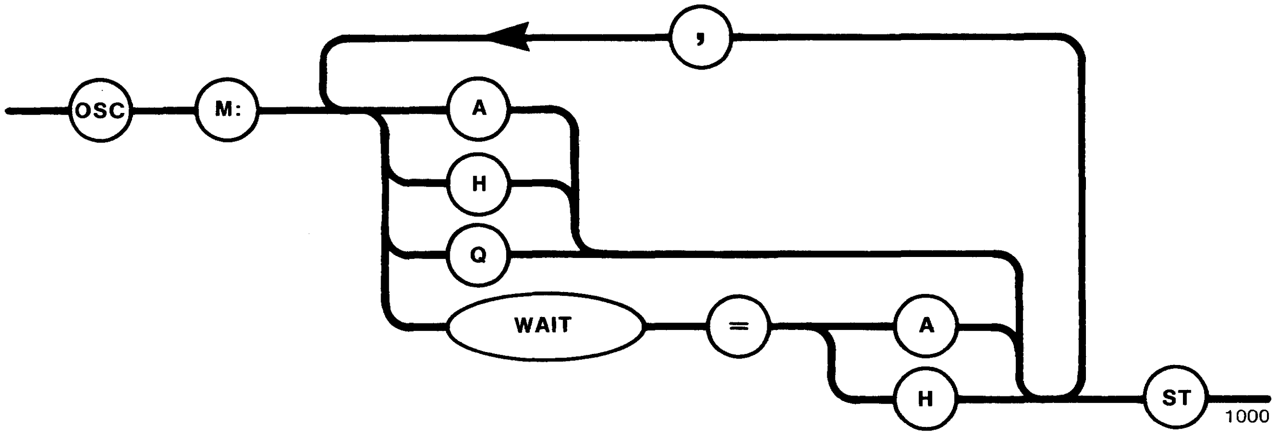
USING A$SPECIAL TO MODIFY CONNECTION AND TERMINAL MODES

As we saw in the previous section, you can use OSC sequences to modify any connection or terminal mode. This brief section is here to remind you that each of these modes (except for the four modes described in Table F-8) can also be changed by means of the A$SPECIAL system call.

USING A MODEM WITH A TERMINAL

The Terminal Support Code supports terminals that interface with an iRMX 86-based application system through a modem. For the most part, tasks and terminals communicate through a modem as if linked by a dedicated line, but they must use OSC sequences to establish a link (dial and answer) and to break the link (hang up).

Before describing the usual protocol for establishing and breaking a modem-based link between a task and a terminal, we need to define the syntax for the necessary OSC sequences. The syntax is as follows:



where:

M	An abbreviation for any word, such as Modem, that begins with that letter.
A	An abbreviation for any word, such as Answer, that begins with that letter.
H	An abbreviation for any word, such as Hangup, that begins with that letter.
Q	An abbreviation for any word, such as Query, that begins with that letter.

Assume that there is a task dedicated to monitoring the modem and performing communication through it. Assume further that the task has a connection to the modem and that the connection is open for both reading and writing. Typical protocol (using the connection) is the following:

1. The task writes the OSC sequence "OSC M:H ST" to the terminal. This means hang up the phone (break the link). This is the initialization step.

2. The task writes the OSC sequence "OSC M:WAIT=A ST" to the terminal and then issues a read request to the terminal. This means that the task wants to be informed when a terminal comes on line and that it will wait for the proper response to return to it by way of the connection.

3. (Eventually) the task receives the APC sequence "APC M:A ST". (Recall that an APC sequence is a message from the Terminal Support Code to a task.) This message means that a terminal user has dialed up the modem and is ready to communicate.

4. The task writes the OSC sequence "OSC M:WAIT=H ST" to the terminal. This causes the Terminal Support Code to send the sequence "APC M:H ST" to the task when the terminal user hangs up.

5. The terminal and the task communicate as if on a dedicated line for as long as is necessary. However, whenever the task receives input, it scans the input for the APC sequence "APC M:H ST".

6. (Eventually) the task receives the sequence "APC M:H ST". This means that the terminal user has hung up and the link is broken.

7. The task returns to step 2.

This protocol is offered as a model and is by no means the only one possible. Note, however, that only the task, and never the terminal, should send OSC sequences to the Terminal Support Code for modem control. This restriction does not apply to other OSC sequences.

Under some circumstances, a task needs to find out whether a terminal is ready to talk to the task via the modem. The task can ascertain the state of the modem (answered or hung up) by performing the following steps, in order:

1. Call A$WRITE to write the sequence "OSC C:T=1,E=1 ST" to the modem. This disables line editing and turns off the echoing to the terminal's screen of characters in the buffer. Note that this is for this connection only, not for other connections (if any) to the modem.

2. Call A$WRITE to write the sequence "OSC M:Q ST" to the modem. This is the Modem Query command, and it is a request for information as to the status of the modem, that is, answered (A) or hung up (H).

3. Call A$READ to read seven characters from the modem. This is a request for an APC sequence of the form "APC M:x ST", where x is A if the modem is answered and H otherwise. This technique will work because the Terminal Support Code places the APC sequence, without a line terminator, at the front of the line buffer for the connection, where data (if any) is awaiting input requests from the task.

After performing these steps, the task can restore the connection's line editing and echo modes to their original states.

THE TERMINAL QUERY COMMAND

The previous section contained information about how to use the Modem Query command. The Terminal Query command is similar, but much broader in scope and purpose, returning the current values of all modes for a terminal and all modes for the connection (to that terminal) through which the command is issued.

A task issues the Terminal Query command by performing the following steps, in order:

1. If desired, call A$WRITE to send an OSC sequence that will set the line editing and echoing modes appropriately. If you want to be able to modify the modes from the terminal, turn on the line editing mode (L=2) and the echoing mode (E=0) for the connection. Otherwise, turn off line editing (L=1) and turn off echoing (E=1).

2. Call A$WRITE to send the OSC sequence "OSC Q ST". This is the Query command, and it causes the Terminal Support Code to place the requested information in the form of an APC sequence (without a line terminator) at the front of the type-ahead buffer for the connection.

3. Call A$READ to read the appropriate number of characters at the connection. This is the part where you have to be careful. The number of characters returned depends on the values of the modes, and some of these modes, such as the input baud rate (I) for the terminal, can vary in length. You should allow two spaces for the APC at the beginning, two spaces for the ST at the end, and enough spaces for the modes in between. A simple, safe way is to read one byte at a time, until "ST" appears. The modes are separated by commas and packed together without blanks. An example is:

```
APC C:T=2,E=0,R=0,W=1,O=0,C=0,T:L=0,H=0,M=0,R=2,W=2,T=1,F=0,I=9600,
O=0,S=18,X=64,Y=24,U=80,V=16 ST
```

The information returned by the Terminal Query command does not include any information about escape sequence-terminal character sequence pairings or about input or output control character assignments.

RESTRICTING THE USE OF A TERMINAL TO ONE CONNECTION

If there are multiple connections to a terminal, any one of the connections can be used to "lock" the terminal. When this happens, the connection used to lock the terminal can be used according to how it was opened, and I/O requests through that connection are processed normally. The connections that are locked out may also be used according to how they were opened, but I/O requests through those connections are queued up until the terminal is unlocked.

An OSC sequence of the form "OSC L ST" locks a terminal, where the L stands for any word that starts with that letter. The command to lock the terminal can come from a task, or it can come from the terminal.

A locked terminal becomes unlocked in either of two ways. One way is for the terminal to be unlocked explicitly, by means of an OSC sequence of the form "OSC U ST", where the U stands for any word beginning with that letter. This command can come from a task, or it can come from the terminal. The other way to unlock the terminal is to close the connection used to lock the terminal.

After a terminal is unlocked, the queued I/O requests are processed in the order in which they were queued.

PROGRAMMATICALLY INSERTING DATA INTO A TERMINAL'S INPUT STREAM

If a task wants to insert data into the input stream at a terminal, it can do so by means of an OSC sequence. After being inserted, the data will appear on the terminal's screen, assuming that the echo mode for the connection used is turned on. Once the data is on the screen, it can be edited, provided that the connection's line edit mode is set for line editing. Being able to do this is useful in cases where it is necessary to enter large blocks of data that vary only slightly from one occurrence to the next.

The form of the OSC sequence that inserts data in a terminal's data stream is "OSC S:text ST"

where:

S	An abbreviation for any word, such as Stuffing, that begins with that letter.
text	The data that is to be placed in the input stream.

COMPOSITE SYNTAX DIAGRAM FOR ALL OSC SEQUENCES

Throughout this appendix, small syntax diagrams for OSC sequences appear, undoubtedly leading you to believe that OSC sequences must be short. However, this is not the case. OSC sequences can be very lengthy, with semicolons separating the pieces that correspond to the diagrams shown earlier in this appendix. With only one exception, different types of OSC sequences can be combined into a larger OSC sequence in any order. The exception is that a composite OSC sequence can contain only one subsequence for inserting data into a terminal's input stream, and that subsequence must be the last subsequence. Figure F-1 shows how the OSC subsequences of this appendix can be combined.

BRIEF REVIEW OF THE USES AND MODIFICATIONS OF CONTROL CHARACTER SEQUENCES

There are three general uses of control characters, and you should be careful not to confuse them. They are the following:

- The Terminal Support Code contains assignments of certain control characters to particular line-editing functions. By means of OSC sequences, these assignments can be altered.
- By means of OSC sequences, control characters can be assigned to trigger translation or simulation.
- By means of the A$SPECIAL system call, control characters can be assigned to be signalling characters.

It is true that you can use control characters for all of these purposes. But be careful in your use of them. If you attempt to dedicate the same control character to different purposes at the same time, the results are unpredictable.

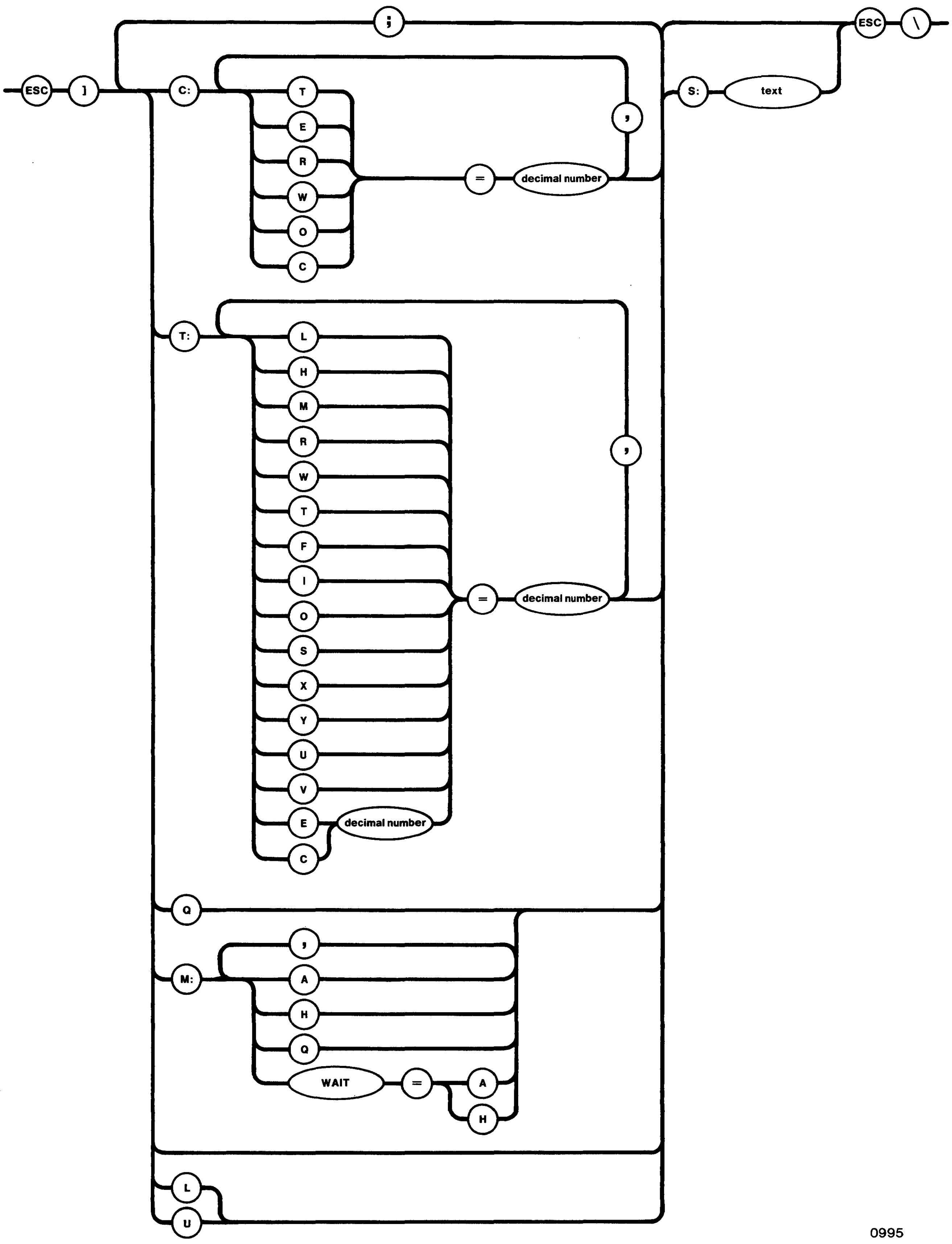


Figure F-1. Composite OSC Sequence Diagram

***

APPENDIX G. INTEL-SUPPLIED TERMINAL DEVICE DRIVERS

Appendix F of this manual describes the full range of I/O capabilities that the Terminal Support Code supports. While it is possible to write a terminal device driver that utilizes all of these capabilities, in practice it is unlikely that a given terminal device driver will require all of them.

Intel Corporation provides, as part of the iRMX 86 product, several device drivers. Among these are terminal drivers that act as interfaces between the Terminal Support Code and the 8251A USART, iSBC 534, iSBC 544, and iSBX 270 devices. The purpose of this appendix is to document the Terminal Support Code capabilities that these drivers support.

THE 8251A USART DRIVER

The USART driver supports an 8251A USART that is connected to any counter of an 8253 Programmable Interval Timer. The hardware that each USART driver supports is restricted to devices which may consist of only one USART.

The only Terminal Support Code feature that the USART driver does not support is modem control. The driver supports baud rate search, with 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200 being the baud rates the driver can recognize. To initiate a baud rate search at the terminal, enter from one to three upper-case U°s. You can easily tell when the baud rate search has been completed, because output to the terminal is held back until then.

THE iSBC® 534 DRIVER

The iSBC 534 driver supports one or more iSBC 534 Four-Channel Communications Expansion boards, each of which has as many as four USARTs. Several iSBC 534 boards may share a single interrupt line, in which case their USARTs are treated as separate units of a single device.

The only Terminal Support Code feature that this driver does not support is separate input and output baud rates. The baud rates that are supported and the manner in which a baud rate search is conducted are exactly as in the case of a USART driver, described in the previous section.

THE iSBC® 544 DRIVER

The iSBC 544 driver supports a number of iSBC 544 Four-Channel Intelligent Communications Expansion boards (or other memory and I/O expansion boards. Several iSBC 544 boards may share a single interrupt line, in which case their channels are treated as separate units of a single device.

The iSBC 544 controller is a self-contained communications processor that incorporates an 8085A CPU for its on-board processing. This on-board processing makes the iSBC 544 driver faster and more efficient than the iSBC 534 driver.

The only Terminal Support Code feature that this driver does not support is separate input and output baud rates. The baud rates that are supported and the manner in which a baud rate search is conducted are exactly as in the case of a USART driver, described in a previous section.

THE iSBX™ 270 DRIVER

The iSBX 270 driver supports one iSBX 270 Video Display Terminal Controller Multimodule board. This board may be mounted either on a CPU board or on a Multimodule board that is directly accessible to the host board°s CPU. The hardware that each iSBX 270 driver supports is restricted to one device, which may consist of only one iSBX 270 controller.

The only Terminal Support Code features that the iSBX 270 driver does not support are modem control, parity checking and setting, and baud rate control, because they are not meaningful in this environment. However, it can strip the parity bit off for input or output using individual connections.

The iSBX 270 controller can be set for any of three display modes. The initial (default) mode is the scrolling mode. The mode can be changed at any time simply by writing an appropriate code to the iSBX 270 controller. The modes and the codes that switch them on are the following:

Code	Mode
0C0H	Scrolling
0C1H	Paging with visible cursor
0C2H	Paging without visible cursor

***

INDEX

Underscored entries are primary references.

***

iRMX™ 86 Basic I/O System
Reference Manual
980312 3-05

REQUEST FOR READER'S COMMENTS

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

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

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

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

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

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

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

NAME ____________ DATE ____________

TITLE ____________

COMPANY NAME/DEPARTMENT ____________

ADDRESS ____________

CITY ____________ STATE ____________ ZIP CODE ____________
(COUNTRY)

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

OMO Technical Publications