Accessing Files Programmer's Guide

900 Series HP 3000 Computer Systems

Manufacturing Part Number: 32650-90885
E0300

U.S.A. March 2000

Notice

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

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Trademark Notice

UNIX is a registered trademark of The Open Group.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

© Copyright 2000 by Hewlett-Packard Company

Contents

1. Introduction

Disk Files and Device Files 15

Topics in this Manual 15

2. Creating A File

The HPFOPEN Intrinsic 18

NOWAIT I/O 18

NOWAIT I/O intrinsics 18

Aborting NOWAIT I/O 19

Limitations 19

The FOPEN Intrinsic 48

The BUILD Command 50

The FILE Command 51

Summary of Overrides 55

Specifying a Record Format 56

Fixed-length records 56

Variable-length records 56

Undefined-length records 57

Specifying a File Type 59

Standard files 59

KSAM files 59

RIO files 59

Circular files 59

Message files 60

Specifying Record Size 61

Specifying Disk Volume Restrictions 63

Specifying a File Code 64

Specifying Storage Format 70

3. Specifying a File Designation

MPE/iX File Designators 72

User-Defined Files 73

Lockwords 74

Backreferencing files 75

System-Defined Files 77

Input/Output sets 78

Passed files 79

Using Command Interpreter Variables and Expressions Within File Designators 83

Parsing and Validating File Designators 84

4. Specifying a File Domain

New Files 87

Temporary Files 87
Permanent Files. 88
Changing Domains 89
Searching File Directories. 90
Listing Files 90

5. Opening a File
How the File System Opens a File 91
Which to Use: HPFOPEN or FOPEN. 93
Opening a Disk File. 93
Opening a new disk file 94
Opening a permanent disk file 95
Opening a System-Defined File 97
Opening $STDIN 97
Opening $STDLIST 98
Opening a Device File 99
Device-dependent file characteristics 99
New and permanent device files 100
Opening an unlabeled magnetic tape file 100
Opening a labeled magnetic tape file 101

6. Closing a File
How the File System Closes a File 103
Closing a Disk File. 105
Closing a new disk file as permanent 105
Closing a permanent disk file 106
Closing a Magnetic Tape File 107

7. Record Selection and Data Transfer
Record Pointers 109
Record Selection. 110
Sequential access 111
Random access 111
Update access 111
RIO access. 113
Multiple Record Transfers 113
Control Operations 114
Spacing 114
Pointing. 114
Rewinding. 115
Magnetic Tape Considerations 115

8. Writing to a File

Sequential Access and Random Access . . . 119
Writing to a disk file using sequential access . . . 120
Writing to a disk file using random access . . . 121
Writing to $STDLIST . . . 122
Writing Messages to the System Console . . . 123
Writing a message to the system console . . . 123
Writing a message to the system console and requesting a reply . . . 124
Writing to a Magnetic Tape File . . . 125
Writing to an unlabeled magnetic tape file . . . 125
Writing to a labeled magnetic tape file . . . 126
Writing a File Label to a Labeled Tape File . . . 127
Writing User Data in ANSI Labels . . . 128

9. Reading from a File

Sequential Access and Random Access . . . 130
Reading from a disk file using sequential access . . . 130
Reading from a disk file using random access . . . 131
Increasing I/O performance using FREADSEEK . . . 132
Reading From $STDIN . . . 133
Reading From a Magnetic Tape File . . . 134
Reading a File Label from a Labeled Tape File . . . 135

10. Updating a File

11. Accessing a File Using Mapped Access

How to Access a File Mapped . . . 142
Advantages of mapped access . . . 143
Short-mapped access . . . 143
Long-mapped access . . . 143
Large-mapped access . . . 144
Opening a File Mapped . . . 144
New Intrinsics . . . 146
HPFADDTOPOINTER . . . 146
HPFFILLDATA . . . 147
HPFMOVEDATA . . . 148
HPFMOVEDATALTOR . . . 149
HPFMOVEDATARTOL . . . 150

12. Sharing a File

Simultaneous Access of Files . . . 153
Exclusive access . . . 154

Semi-exclusive access 155
Shared access 155
Multiaccess 156
Global multiaccess 156
Sharing the File Using FLOCK and FUNLOCK 157

13. Maintaining File Security
Access Control Definition Security (ACD) 159
ACD scope 159
Owners 159
How acds work 160
ACD modes 160
Managing ACDs with commands and intrinsics 161
Preserving ACDs 162
Managing ACDs 162
Logging system events 167
Logging a specific user 174
Logging file security related events. 175
Traditional Mechanism for File Security 179
Specifying and restricting file access by access mode 179
Specifying and restricting file access by type or user 181
Changing security provisions of disk files. 185
Suspending and restoring security provisions 186

14. Getting File Information
Displaying General File Information 188
Displaying permanent file information with LISTFILE 188
Displaying temporary file information with LISTFILE...(;TEMP) 193
Displaying file equations with LISTEQ 193
Retrieving Specific File Information. 194
[:CMD] FINFO 194
FFILEINFO 195
FGETINFO 196
FLABELINFO 197
Determining Interactive/Duplicative Files with FRELATE 197
Displaying File Error Information 198
FCHECK 198
FERRMSG 199
PRINTFILEINFO. 199
Writing a file system error-check procedure 201

A. Pascal/XL Program Examples

Program Example A-2 208
Program Algorithm 208
Source code listing 209
Program Example A-3 217
Program Algorithm 217
Source code listing. 217
Program Example A-4 221
Program Algorithm 221
Source code listing. 222
Program Example A-5 224
Program Algorithm 225
Source code listing 225

Figures

Figure 1-1.. File System Interface 13
Figure 1-2.. Records/Files Relationship 14
Figure 2-1.. File System Hierarchy of Overrides 55
Figure 2-2.. Fixed-Length Records 56
Figure 2-3.. Variable-length Records 57
Figure 2-4.. Undefined-Length Records 58
Figure 2-5.. Record Placement for ASCII Files 62
Figure 3-1.. Passing Files between Program Runs 80
Figure 3-2.. Passing Files within a Program Run 81
Figure 3-3.. Illustration of FPARSE Usage 84
Figure 3-4.. Illustration of FPARSE Usage 85
Figure 3-5.. Illustration of FPARSE Usage 85
Figure 5-1.. File System Hierarchy of Overrides 92
Figure 6-1.. Using the FCLOSE Intrinsic with Unlabeled Magnetic Tape 107
Figure 7-1.. Record Pointers 110
Figure 7-2.. Magnetic Tape Markers 116
Figure 12-1.. Requested Access Granted, Unless Noted 155

Tables

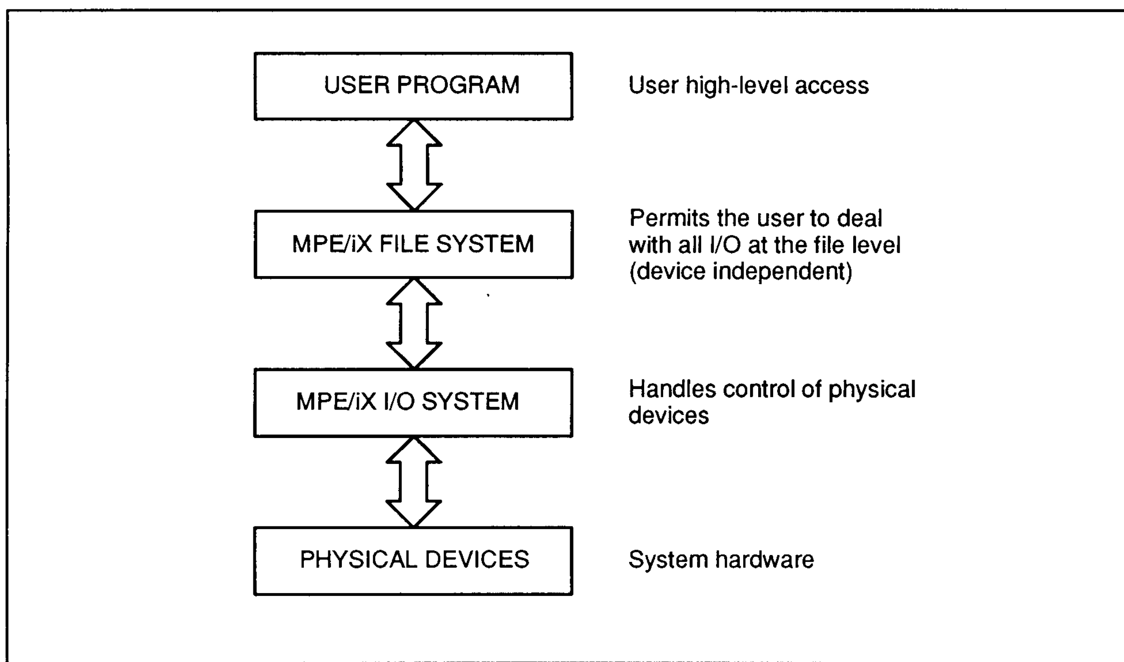
Table 2-1.. FOPEN/HPFOPEN Parameter Equivalents 48
Table 2-2.. Determining a File's Physical Characteristics Using FOPEN 49
Table 2-3.. FILE, FOPEN, and HPFOPEN Parameters 52
Table 2-4.. Comparison of Logical Record Formats 58
Table 2-5.. Standard Default Record Sizes. 62
Table 2-6.. Reserved File Codes 64
Table 3-1.. System-Defined File Designators. 77
Table 3-2.. Input Set 78
Table 3-3.. Output Set 78
Table 3-4.. New Files Versus $NEWPASS 81
Table 3-5.. Old Files Versus $OLDPASS 82
Table 4-1.. Features of New, Temporary, and Permanent Files 88
Table 4-2.. File Domains Permitted 88
Table 5-1.. Device-Dependent Restrictions 99
Table 7-1.. Intrinsics for Data Transfer 111
Table 12-1.. File Sharing Restriction Options 154
Table 13-1.. SYSGEN System Logging. 167
Table 13-2.. Type 135 Record Format. 169
Table 13-3.. Type 136 Record Format. 169
Table 13-4.. Type 137 Record 170
Table 13-5.. Type 139 Record 172
Table 13-6.. Type 140 Record Format. 173
Table 13-7.. Type 141 Record 173
Table 13-8.. Type 134 Record Format. 177
Table 13-9.. Type 138 Record Format. 178
Table 13-10.. Traditional File Access Mode Types. 180
Table 13-11.. Effects of Access Modes. 181
Table 13-12.. User Type Definitions (Traditional Security) 181
Table 13-13.. Default Security Provisions (Traditional) 185
Table 14-1.. Format Selection 189
Table 14-2.. FINFO Options 194
Table 14-3.. PRINTFILEINFO Information 200

1 Introduction

Almost every kind of organization in our modern society is concerned in some way with information. Corporations keep track of their business dealings, political groups keep lists of potential voters, and families remember whose turn it is to do the dishes. When an organization needs to deal with large amounts of information in an efficient, dependable manner, a computer can be an indispensable aid. This manual describes the MPE/iX file system that is responsible for handling information in the 900 Series of the HP 3000 Family.

Figure 1-1. shows the relationships among your program, the MPE/iX file system, the MPE/iX I/O System, and the actual hardware of the system. Notice that the MPE/iX file system serves as the interface between you and the rest of the system.

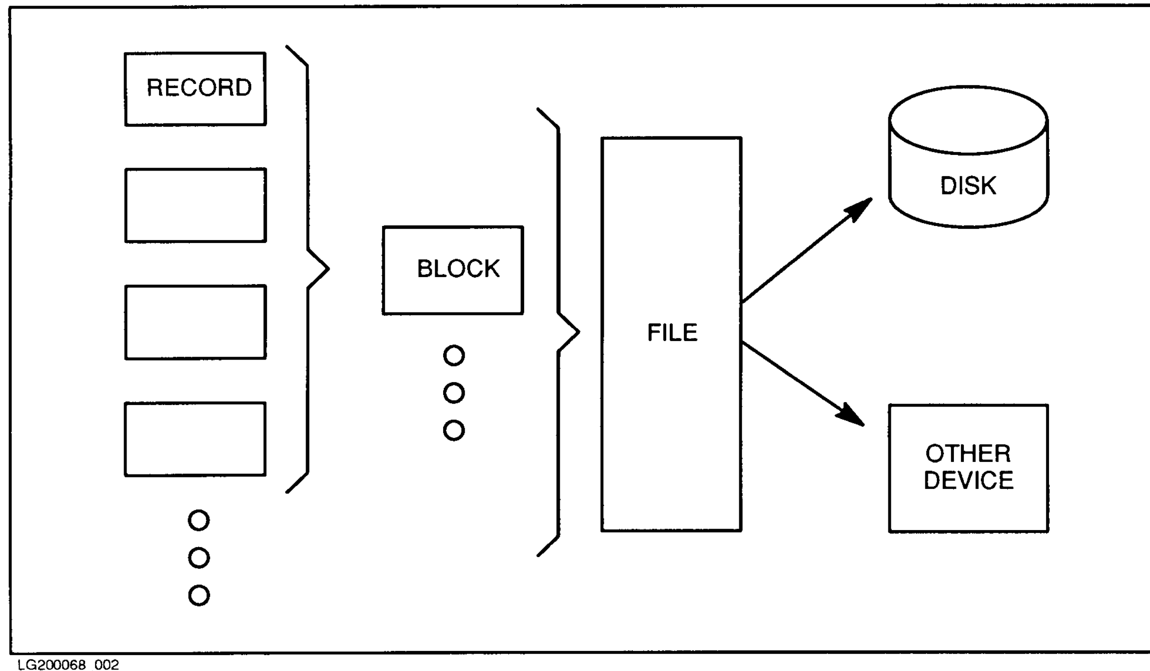
Figure 1-1. File System Interface



The file system is the part of the MPE/iX operating system that manages information being transferred or stored with peripheral devices. It handles various input/output operations, such as the passing of information to and from user processes, compilers, and data management subsystems. Conceptually, these data transfers are very simple: information is arranged into data elements within a record; this record is then input,

processed, and output as a single unit. Logically related records are grouped into sets known to the file system as files, which may be kept in any storage medium or sent to any input/output peripheral (as illustrated in Figure 1-2. below).

Figure 1-2. Records/Files Relationship



Since all input/output operations are done through the mechanism of files, you may access very different devices in a standard, consistent way. It does not make much difference to you whether you read your file from a disk or from a magnetic tape, because the file system permits you to treat all files in the same way. This property of the file system gives your program device independence; the name and characteristics assigned to a file when it is defined in a program do not restrict that file to residing on the same device every time the program is run. You, the user, need only reference the file by the file name assigned to it when it was created, and the file system determines the device or disk address where the file is stored and access the file for you. (Of course, you should be aware of the properties of the device you're using. For example, the MPE/iX file system permits you to read a file from a line printer.)

Disk Files and Device Files

The file system recognizes two basic types of files, classified on the basis of the media on which they reside when processed:

1. *Disk files*, which are files residing on disk, are immediately accessible by the system and potentially shareable by several jobs/sessions at the same time.
2. *Device files* are files currently being input to or output from any peripheral device except a disk. When information exists on such a device but is not being processed, the file system cannot recognize it as a file. Thus, information on a magnetic tape is not identified as a file until the tape is loaded onto a tape drive and reading begins; data being written to a line printer is no longer regarded as a file when output to the printer terminates. A device file is considered nonshareable; it is accessed exclusively by the job or session that acquires it, and is owned by that job/session until the job/session explicitly releases it or terminates.

NOTE Spooled device files, although temporarily residing on disk, are considered device files in the fullest sense because they are always originated on or destined for devices other than disk, and because you generally remain unaware of their storage on disk as an intermediate step in the spooling process. Whether they deal with spooled or unspooled device files, your programs handle input/output as if the files reside on nonshareable devices. The console operator, not the user, controls the spooling operation.

Topics in this Manual

When you create a file, you specify certain permanent attributes that the file will have based upon its intended use. Chapter 2, "Creating a File" describes the physical characteristics that are determined when you create a file and the intrinsics and commands you use to specify those physical characteristics

What name do you give a file that you create? How does MPE/iX recognize your file? Chapter 3, "Specifying a File Designator" describes how you designate a file name in your program and discusses file naming conventions maintained by MPE/iX.

You may classify your file as a new, temporary, or permanent file. Chapter 4, "Specifying a Domain" discusses these classifications.

Before your program can access or otherwise manipulate a file, the program must open that file. Program examples illustrating various ways you can open a file are described in Chapter 5, "Opening a File."

Once your program is finished accessing or manipulating a file,the file must be closed. Program examples illustrating various ways you can close a file are described in Chapter 6, "Closing a File."

One of the file system's principal concerns is the transfer of information to and from your files. Chapter 7, "Record Selection and Data Transfer" discusses how you can use MPE/iX file system intrinsics to select records and transfer data between your program and files.

Program examples illustrating various ways to write data from your program to a file are described in chapter 8, "Writing to a File."

Program examples illustrating various ways to read data from a file are described in chapter 9, "Reading From a File."

A special method of accessing a file, called **update access**, is discussed in chapter 10, "Updating a File."

You can access a file mapped directly through memory loads and stores, thus bypassing the overhead associated with accessing the file through file system intrinsics. Chapter 11, "Accessing a File Using Mapped Access" describes mapped access of a file and the applications where mapped access may increase your program's performance.

There are special considerations you must take into account when you are accessing a file that is being shared concurrently by others. Chapter 12 ,"Sharing a File" discusses file sharing methods available to you.

Associated with each account, group, and individual file, is a set of security provisions that specifies any restrictions on files in that account or group, or to that particular file. These provisions are discussed in chapter 13, "Maintaining File Security."

MPE/iX provides a number of commands and intrinsics that enable you to obtain information about your files. You can use the commands and intrinsics described in chapter 14, "Getting File Information" for a variety of purposes.

The HP Pascal/iX program examples found in appendix A, "HP Pascal/XL Program Examples" are provided to help you better understand how to use file system intrinsics to perform file access tasks.

2 Creating A File

When you create a file, you choose the attributes that file will have; your choices are made on the basis of how the file will be used. A file's physical characteristics are determined by the parameters you choose when you create the file with the `HPFOPEN/FOPEN` intrinsic or the `BUILD` command, or when you specify the file with the `FILE` command.

Once a file has been created, its physical characteristics cannot be changed. The file can be renamed or purged, but the only way to change its physical characteristics is by building a new file and copying the contents of the old file into the new.

File equations and `HPFOPEN/FOPEN` calls cannot alter physical characteristics of an existing file, but they can alter the way the file is to be used. Other characteristics of the file that you create can be redefined each time you open the file. Those characteristics are discussed in later chapters in this manual.

In this chapter, we will address the following questions:

- What intrinsics and commands can you use to specify a file's physical characteristics?
- What are the physical characteristics that are determined when you create a file?

The HPFOPEN Intrinsic

The HPFOPEN intrinsic establishes access to a disk or device file and enables you to create a file on a shareable device. HPFOPEN is used to define a file's physical characteristics, including file and record structure. Its syntax is

```
HPFOPEN (filenum, status,  [, itemnum, item]...);
```

The HPFOPEN optional parameters are a superset of the options provided in the FOPEN intrinsic and provide more efficient access to files.

NOWAIT I/O

Sometimes a programmer wants an application to read or write a record, but does not want it to wait for I/O to complete. For such an application, waiting is wasting time when it could be doing other processing. Timeouts do not adequately address this problem. The programmer wants this application to start an I/O, continue processing immediately, and check periodically to see if the I/O has finished

MPE/iX provides a way to solve this problem with NOWAIT I/O. This feature is requested by enabling the NOWAIT I/O option (item #16) in HPFOPEN.

When using NOWAIT I/O, the process must make at least two intrinsic calls to perform the I/O, one to start it and one to finish it. MPE/iX still handles the file in the same way; but instead of waiting for the I/O to complete, MPE/iX returns control to the application so that the application can do some useful processing.

NOWAIT I/O has been available to users of standard files for a long time, but to use it on standard files requires privileged mode. On standard files the mechanics of NOWAIT I/O prevent MPE/iX from protecting a process from corrupting its own stack; however, because message files work differently, NOWAIT I/O on message files does *not* require privileged mode.

NOWAIT I/O intrinsics

To perform a NOWAIT I/O, the FREAD or FWRITE intrinsic must be called to initiate the transfer. These intrinsics return immediately, and no data is transferred yet. The return value for FREAD is set to zero and is not needed. To complete the transfer, either IODONTWAIT or IOWAIT must be called. IODONTWAIT tests whether the I/O has finished. If it has, the intrinsic returns a condition code of CCE and the file number as the return value. If the I/O has not completed, CCE and a zero return value are passed back. If IOWAIT is called, it waits until the I/O has finished, like a normal WAIT I/O FREAD or FWRITE.

Only one NOWAIT I/O can be outstanding against a file by a particular accessor at a time; however, an accessor can have NOWAIT I/Os outstanding against several files at the same time. These I/Os can be completed by a "generalized" IODONTWAIT or IOWAIT: the file number parameter is zero or is omitted. In this case, these intrinsics report on the first I/O to complete, returning the file number for that file. If the call to one of these intrinsics is in a loop, then that one call can be used to complete all the NOWAIT I/Os.

Aborting NOWAIT I/O

Occasionally, after a process has started a NOWAIT I/O with `FREAD` or `FWRITE`, something occurs that causes completion of that I/O to be no longer needed. Perhaps the process is "shutting down" and does not want to wait for the I/O (that is, to issue `IOWAIT` or `IODONTWAIT`).

MPE/iX lets the process abort NOWAIT I/Os that have not yet completed by using `FCONTROL` with a control code of 43. A condition code of CCE is returned if the I/O was aborted; in this case, nothing more needs to be done. CCG is returned if the I/O has already completed; in this case, `IODONTWAIT` or `IOWAIT` must be called to clear it. CCL and `FSERR 79, No NOWAIT I/O pending for special file` are returned if there was nothing to abort.

Limitations

Currently, MPE/iX does not support NOWAIT I/O to message files across a network. In most cases, this is not an important limitation, because it is rare that both readers and writers to the same message file need to use NOWAIT I/O. If the file is made local to the accessor that needs NOWAIT I/O, the other accessor can then do WAIT I/O across the network.

More information on these intrinsics is found in the *MPE/iX Intrinsics Reference Manual*. For detailed information about WAIT and NOWAIT, consult the *Interprocess Communications Programmers' Guide*.

The following lists the optional parameters you can use to specify a file's physical characteristics, as well as the default values for each.

Itemnum/ Mnemonic	Item Description
0	End of option list:
	There is no corresponding *`item`*. The absence of an *`itemnum`* after the last *`itemnum`*,*`item`* pair is equivalent to specifying this option.
2/CA	Formal designator:
	Passes a formal file designator that is interpreted according to MPE-escaped semantics (unless another syntax has been chosen via item 41). The first charater is interpreted as a delimiter, and all subsequent characters, up to the next occurrence of the delimiter, comprise the formal designator. The file name must be terminated by a nonalphanumeric character other than a period (.), a slash (/), a hyphen (-), and an underscore (_). Use of matched starting and ending name delimiters (a quoted name) alleviates the need for a terminating character other than the quote characters.
	The file referred to by *`formaldesig`* can be either an MPE file (i.e., one that uses MPE syntax) or it can follow HFS syntax. If *`formaldesig`* follows MPE syntax, the file name can include password, group, and account specifications. The file name can backreference a file equation and optionally be preceded by an asterisk. If *`formaldesig`* follows HFS syntax,

the file name must start with either a dot (.) or a slash (/).

The file referred to by *formaldesig* may reside either in an MPE group or in an HFS directory. For files located in HFS directories, traverse directory entries (TD) access is required to all directories specified in *formaldesig*. If there is no TD access, HPFOPEN fails and a value of -180 is returned in the *status.info* parameter. If you are using HPFOPEN to create a file or hierarchical directory and you lack of create directory entry (CD) access, *status.info* will return a value of -179.

If *formaldesig* is an escaped pathname:

- you cannot reference remote files
- it cannot express a name equivalent to *filename*:*envid*
- you cannot use the *device* parameter (*device=node#*) to specify the remote location of a device

If *formaldesig* is the name of a user-defined file, it can begin with an asterisk (*). If *formaldesig* is the name of a system-defined file, it can begin with a dollar sign ($). When creating a KSAM file, *formaldesig* must be a unique file name, that is, one not currently existing in the permanent file directory.

The formal file designator can contain command interpreter variables and expressions that are evaluated before *formaldesig* is parsed and validated.

As the default, HPFOPEN creates a nameless file that can be read or written to, but not saved. (The domain option of a nameless file must specify a new file unless it is a device file.)

The following are examples of valid formal file designators:

```
&file/lock.group.account:node.dest.level&

&filename&

&!myfile&

&!afile/![FINFO("!afile",33)]&
```

The following are examples of invalid formal file designators:

`"filename.group` (missing delimiter ("))

`file.group"` (`'f'` is used as delimiter, missing at end)

(ASC) It is recommended that this *itemnum*, *item* pair be used for asynchronous devices.

When you use HPFOPEN to open a file, you may use either *itemnum*=2 or *itemnum*=51; you cannot use both.

3/I32 Domain:

Passes a value indicating which file domain the system searches to locate the file. A nameless disk file must always be a new file.

A device file (such as a tape drive, terminal, spooled printer or *hot* printer) always resides in the system file domain (permanent file directory). Always specify a device file as old or permanent.

The following values are valid:

0	The file is a new temporary file. It is not placed in a directory.
1	The file is a permanent file, found in the system file domain.
2	The file is a temporary file, found in the job file domain.
3	The file is an old (permanent or temporary) file. The job file domain is searched first. If the file is not found, the system file domain is searched.
4	The file is created, placed in the permanent file directory, and becomes a permanent file.

Hierarchical directories must be created in the permanent file domain by specifying the create file domain (4).

Default: 0

5/I32 Designator:

Passes a value indicating a special file opening. Any of the following special files can be specified with the *itemnum*=2. For example, a file name of `$STDLIST` opens the standard list device. The following values are valid:

0	Allows all other options to specify the file.
1	The actual file designator is `$STDLIST`.
2	The actual file designator is `$NEWPASS`.
3	The actual file designator is `$OLDPASS`.
4	The actual file designator is `$STDIN`.
5	The actual file designator is `$STDINX`.
6	The actual file designator is `$NULL`.

Default: 0

For example, if `MYFILE` is passed in *itemnum*=2, then using *itemnum*=5 and *item*=4 to equate it with `$STDIN` is equivalent to the file equation `FILE MYFILE=$STDIN`.

This option is not equated with *itemnum*=2 if both of the following conditions are true:

- The *itemnum*=9 option allows file equations for the file opening.
- An explicit or implicit `FILE` command equating the formal file designator to a different actual file designator occurs in the job/session.

A leading * in a formal file designator passed by *itemnum*=2 overrides an

itemnum=9 option.

6/I32 Record format:

Passes a value indicating the internal record structure desired for the file. This option is applicable only at file creation.

The following values are valid for records:

0	Fixed-length
1	Variable-length
2	Undefined-length (no implied structure)
9	Byte stream
10	Hierarchical directory

Default: 0

Byte stream record format may be specified only for standard disk files (*itemnum* 10 equal to 0). Hierarchical directory record format is the the default record format when creatinng a directory (*itemnum* 10 equal to 9). *Itemnum* 10 equal to 9 is the only record format which may be specified when creating a directory. Hierarchical record format is only specified for the directory file type. Record formats not implemented for the specified file type are ignored. Byte stream and hierarchical directory record formats are supported only on disk devices.

(ASC) This *itemnum,item* pair is ignored for files opened on a terminal; records of files on terminals are of undefined length. If the file is to be redirected to tape or disk, set the value to 0 (fixed-length).

FIFO file must be created with the byte streams (9) record format.

7/I32 Carriage-control:

Passes a value indicating whether or not a carriage-control directive is supplied in the calling sequence of each FWRITE call that writes records onto the file. This option is applicable only at file creation.

The following values are valid:

0	No carriage-control directive expected
1	Carriage-control directive expected

Default: 0

Carriage-control is defined only for ASCII files. This option and *itemnum*=53 are exclusive, and attempts to open new files with both binary and carriage-control directives result in an access violation.

A carriage-control character passed through the *control* parameter of FWRITE is recognized for files with carriage-control specified in HPFOPEN/FOPEN. Embedded control characters are treated as data on files where no carriage-control is specified, and spacing is not invoked for the file. Specify spacing action on files where carriage-control has been specified by either embedding the control in the record, indicated with a

control parameter in the call to FWRITE, or by sending the control code directly through the *control* parameter of FWRITE.

If a carriage-control character is sent to a file where the control cannot be executed directly (for example, line spacing characters sent to a disk or tape file), the control character is embedded as the first byte of the record. Therefore, the first byte of each record in a disk file having carriage-control characters enabled contains control information. If carriage-control characters are sent to other types of files, the control is transmitted to the driver.

Control codes %400 through %403 are remapped to %100 through %103, so that they fit into one byte and can be embedded. Records written to the line printer with control codes %400 through %403 should contain only control information.

Records written with control codes %400 through %403 and no data (count=0, or embedded control and count=1) does not cause physical I/O.

For computing record size, the file system considers carriage-control information as part of the data record. Therefore, specifying the carriage-control option adds one byte to the record size when the file is originally created. For example, a specification of REC=-132,1,F,ASCII;CCTL results in a *recsize* of 133 bytes.

Generally, the entire record can be read. Refer to the table listing the item values returned by the FFILEINFO intrinsic. However, on writes to files where carriage-control characters are specified, the data transferred is limited to *recsize*-1 unless a control of one is passed, indicating the data record is prefixed with embedded carriage-control characters.

The value of this *itemnum* is ignored when a byte stream or hierarchical directory is created. Byte stream files and hierarchical directories are created without carriage control (NOCCTL).

8/CA — Enable tape label:

Passes the tape label name of a labeled tape. The name must follow the ANSI standards for tape label names. The name consists of <=6 printable characters that identify the volume. In a multivolume set, only the first tape label can be specified.

Default: a null tape label

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example:

%volid% (% is the delimiter, *volid* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

(ASC) Not used for asynchronous devices.

9/I32 — Disallow file equation:

Passes a value indicating whether or not file equations are allowed. A leading `*` in a formal file designator overrides the setting to disallow file equations.

The following values are valid:

0	Allow file equations to override programmatic or system-defined file specifications.
1	Disallow file equations from overriding programmatic or system-defined file specifications.

File equations can be enabled for escaped pathnames expressed using MPE-escaped name semantics or names expressed using POSIX name semantics, but a matching file equation is not found since the file designator on the left side of a file equation can only be expressed using MPE-only syntax.

Default: 0

10/I32 File type:

Passes a value indicating the internal record structure used to access records in the file. If the file is old, this option is ignored. Specifying an *itemnum*=5 value other than zero overrides this option. This option is applicable only at file creation.

The following values are valid:

0	Standard (STD) file
1	KSAM/3000 file
2	Relative I/O (RIO) file
3	KSAM XL file
5	NM spoolfile
4	Circular (CIR) file
6	Message (MSG) file
7	KSAM 64 file
9	Directory

Default: 0

Hierarchical directories must be created in the permanent file domain.

KSAM/3000 (1), RIO (2) and CIR (4) file types may only be created using names belonging to the MPE name space.

(ASC) Set the value to 0 for asynchronous devices.

11/I32 Access type:

Passes a value indicating the type of access intended for the file. This option restricts/allows usage of the file system intrinsics.

The following values are valid:

0	Read access only, if the file's security provisions allow read access. FWRITE, FUPDATE, and FWRITEDIR intrinsic calls cannot reference this file. The end-of-file (EOF) is not changed; the record pointer starts at zero. (Default)
1	Write access only, if the file's security provisions allow write access. Any data written in the file prior to the current HPFOPEN request is deleted. FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file. The EOF is set to zero; the record pointer starts at zero. On magnetic tape an EOF is written to the tape when the file is closed even if no data is written.
2	Write-Save access only, if the file's security provisions allow write access. Previous data in the file is not deleted. FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file. The EOF is not changed; the record pointer starts at zero. Therefore, data is overwritten if FWRITE is called. The system changes this value to append for message files.
3	Append access only, if the file's security provisions allow either append or write access. FREAD, FREADDIR, FREADSEEK, FUPDATE, FSPACE, FPOINT, and FWRITEDIR intrinsic calls cannot reference this file. The record pointer is set to EOF prior to each FWRITE. For disk files, the EOF is updated after each FWRITE call. Therefore, data cannot be overwritten.
4	Read/Write (I/O) access only, if the file's security provisions allow both read and write access. If both read and write access are not allowed, the access type is limited to that specified in the security provisions (either read or write). Any file intrinsic can be issued except FUPDATE for this file. The EOF is not changed; the record pointer starts at zero. This option is not valid for message files.
5	Update access only, if the file's security provisions allow both read and write access. If both read and write access are not allowed, the access type is limited to that specified in the security provisions (either read or write). All file intrinsics can be issued, including FUPDATE, for this file. The EOF is not changed; the record pointer starts at zero. This option is not valid for message files.
6	Execute access only, if the file's security provisions allow execute access. This allows read/write access to any loaded file. The program must be running in privileged mode to specify execute access. This option is not valid for message files.
7	Execute-Read access only, if the file's security provisions allow execute access. This allows only read access to a

loaded file. The program must be running in PM to specify execute-read access. This is changed to execute access for KSAM, CIR, and RIO files. Not valid for message files.

8 — Reserved for MPE/iX. No access, opens the file or directory without any access checking. A process must be executing in system code to use this access type.

9 — Directory read access, opens a directory for directory read access. Directories can only be opened for no access or directory read access. Files cannot be opened for directory read access.

FIFO files should be opened for Read Access Only (0) or Write Access Only (1). Other access types can cause unexpected results to occur.

12/I32 — Dynamic locking:

Passes a value enabling/disabling file locking for the file. When specified, the `FLOCK` and `FUNLOCK` intrinsics can be used to dynamically permit/restrict concurrent access to a disk file by other processes at specified times.

The following values are valid:

0 — Disallow dynamic locking/unlocking

1 — Allow dynamic locking/unlocking

Default: 0

The process can continue this temporary locking/unlocking until it closes the file. If several accessors are sharing the file, they must all specify, or not specify, this option. For example, if a file is opened with the dynamic locking option enabled, and a subsequent accessor tries to open the file with dynamic locking disabled, that subsequent attempt to open fails.

Dynamic locking/unlocking is possible through the equivalent of a global resource identification number (RIN) assigned to the file and temporarily acquired by `HPFOPEN`.

Accessors that have opened a file with the dynamic locking option enabled must access the file through the `FLOCK` and `FUNLOCK` intrinsics to gain exclusive access to the file. Since the use of these intrinsics is discretionary, however, all accessors must agree to use `FLOCK` and `FUNLOCK` when writing to a file to guarantee exclusive access. File locking is advised, but is not mandated by MPE/iX.

Lock access must be at the account, group, and file levels for `HPFOPEN` to grant this option. (Lock access is available if lock, execute, append, or write access is set at these levels.) This option is ignored for files not residing on disk.

This *`itemnum`* may only be specified with the disallow dynamic locking value (0) when used with directories.

(ASC) Not used for asynchronous devices.

13/I32 Exclusive:

Passes a value indicating continuous exclusive access to the file, from open to close. Use this option when performing a critical operation (for example, updating the file).

The following values are valid:

0 — If *itemnum*=11 specifies read only access, read-share access takes effect. Otherwise, exclusive access takes effect. Regardless of which access option was selected, FFILEINFO reports zero. A zero (default) value for the *itemnum* specifies that if the access type is read, directory read, or no access (*itemnum* 11 equal to 0, 8, or 9) then shared access takes effect; otherwise exclusive access takes effect.

1 — Exclusive access. After the file is opened, any additional HPFOPEN/FOPEN requests for this file, whether issued by this process or another process, are prohibited until this process issues the FCLOSE request or terminates. If any process is already accessing this file when an HPFOPEN/FOPEN call is issued with exclusive access specified, an error status is returned to the process. If another HPFOPEN/FOPEN call is issued for this file while exclusive access is in effect, an error code is returned to the process that issued that HPFOPEN/FOPEN call. Request exclusive access only if the lock access mode is allowed by the security provisions for the file. For message files, specifying this value means that there can be only one reader and one writer.

Exclusive access cannot be used with directories.

2 — Read-Share access (semi-exclusive access). After the file is opened, concurrent write access to this file through another HPFOPEN/FOPEN request is prohibited, whether issued by this process or another process, until this process issues the FCLOSE request or terminates. A subsequent request for the read/write or update *itemnum*=11 obtains read access. However, other types of read access are allowed. If a process already has write access to the file when this HPFOPEN call is issued, an error code is returned to the calling process. If another HPFOPEN/FOPEN call that violates the read-only restriction is issued while read-share access is in effect, that call fails and an error code is returned to the calling process. You can request read-share access only if you are allowed the lock access mode by the security provisions for the file. For message files, specifying this value means that there can be multiple readers, but only one writer.

3 — Share access. After the file is opened, this permits

		concurrent access to this file by any process, in any access mode, subject to other basic security provisions in effect. For message files, specifying this value means that there can be multiple writers and one reader.

Default: 0

(ASC) This option is ignored for devices.

FIFO files should be opened for Share Access (3). Other exclusive accesses to file will cause unexpected results to occur.

14/I32 Multiaccess:

Passes a value indicating how the file's record pointer is to be shared. This option is useful for sharing standard input devices where there is some natural sequence of access to the file. This option permits processes located in different jobs or sessions to open the same file and share that file's record pointer.

The following values are valid:

0	No multiple process access allowed. A unique record pointer is created for this access to the file. For message files, the file system sets the multiaccess option to 2 when a zero is specified for this option.
1	Intrajob multiprocess access allowed. A record pointer is shared with all other opened files of the same name in the same job/session who opened the file with `itemnum`=14 is set to either 1 or 2.
2	Interjob multiprocess access allowed. A record pointer is shared with all other opened files of the same name on the system. This is the same as specifying the `GMULTI` option in a `FILE` command.

Default: 0

Native byte stream access (see item 77) is opened regardless of the value of this *itemnum*. This *itemnum* is also ignored for directories since it is not applicable.

(ASC) Not used for asynchronous devices.

15/I32 Multirecord:

Passes a value indicating that individual read or write requests are not confined to record boundaries.

The following values are valid:

0	Nonmultirecord mode (`NOMULTI`)
1	Multirecord mode (`MULTI`)

Default: 0

If the number of half words or bytes to be transferred (specified in the

length parameter of the read or write request) exceeds the size of the physical record (that is, a block) that is referenced, the remaining half words or bytes are taken from subsequent successive records until the number specified by *length* has been transferred. For message (MSG) files not accessed with *itemnum*=17 enabled, the file system sets this option to zero. This option is available only if *itemnum*=46 is set to 1.

(ASC) This option is not used for printers.

16/I32 Nowait I/O:

Allows the accessor to initiate an I/O request and to have control returned before the completion of the I/O. This option implies the inhibit buffering option; if NOBUF is not specified, the file system does it. Multirecord access is not available. This option is not available if the file is located on a remote computer. When opening nonmessage files, the process must be running in PM (execution level 2) to specify this option. Set *itemnum*=29 to 3 if the file is to be accessed while in user mode (execution level 3).

The following values are valid:

0 Nowait I/O not in effect

1 Nowait I/O in effect

Default: 0

Directories may not be opened using Nowait I/O (1).

17/I32 Copy mode:

Passes a value that determines if any file should be treated as a standard sequential file so it can be copied by logical record or physical block to another file.

Byte stream files and directories are accessed using normal access (0) regardless of the value specified for this *itemnum*.

The following values are valid:

0 The file is accessed as its own file type (for example, a message file is treated as a message file).

1 The file is to be treated as a standard (STD) file with variable-length records. For message files, this allows nondestructive reading of an old message file at either the logical record or physical block record level. Only block-level access is permitted if the file is opened with write access. This prevents incorrectly formatted data from being written to the message file while it is unprotected. To access a message file in copy mode, a process must have exclusive access to the file.

(KSAM) only allowed for read access for KSAM XL or KSAM64 files.

Default: 0

18/I32 Short-mapped:

Returns a short pointer to the beginning of the data area of the file. This option maps the file into short pointer space. A short-mapped file can be 4 megabytes in length. The calling process can have up to 6 megabytes of short-mapped files open at a time. Use the pointer as a large array of any type to efficiently access the file.

NOTE SHORT MAPPED files are limited to 4 MB in size per file with a 6MB/process limit. The system space that holds all SHORT MAPPED files is limited to 1GB with some space already used by the operating system. If a large number of files or large amount of space is needed, consider opening the files LONG-MAPPED (item number 21) or LARGE-MAPPED (item number 87)

A file previously opened normally (not mapped) or with the long-mapped option is not accessible with the short-mapped option. If this option is specified with the file already opened into long pointer space, an error results.

A loaded program file or a loaded library file is not accessible with the short-mapped option. A file cannot be loaded that is currently opened with the short-mapped option.

Sharing of short pointer files is provided through normal file system sharing mechanisms, for example, use of the exclusive option. With the short-mapped file, all file system intrinsics, applicable to the file, can be used. `FREAD` and `FWRITE` calls can be mixed with the short-mapped access.

Standard (STD) type disk files of fixed or undefined record length can be accessed short-mapped with the access type option set to any value. Standard type disk files of variable record length can be accessed short-mapped only if the access type option is set to read-only access. KSAM files can be accessed short-mapped only if the access type option is set to read-only access and the copy mode option is set to 1.

Directories may not be opened using short-mapped access.

Default: No short pointer returned

(ASC) Not used for asynchronous devices.

19/I32 Record size:

Passes the size, in bytes, of the logical records in the file. Valid range is dependent upon both storage format (ASCII or binary) and record format. For fixed-length and undefined-length ASCII files, a record size can be specified in the range 1..32,767. For variable-length ASCII files, and for fixed-length, variable-length, and undefined-length binary files, a record size can be specified in the range 1..32,766.

`HPFOPEN` rounds up odd values to the next highest even number (equivalent to the nearest half word boundary) if the file is ASCII with variable-length record format, binary with fixed-length, variable-length, or undefined-length record format.

For example, if a record size of 105 is specified for a fixed-length binary file, HPFOPEN sets the record size to 106; if a record size of 233 is specified for a fixed-length ASCII file, the record size remains the same as it was when specified.

The value specified for this *itemnum* is ignored when a byte stream file or hierarchical directory is created. Byte stream files are created with a logical record size of one byte (1). Hierarchical directories are created with a logical record size of 32 bytes (32).

Default: 256

(ASC) For terminal and printer files, no rounding up occurs if a record size consisting of an odd number of bytes is specified. The record size can be different from the port configuration. The default is the configured record size (normally 40 words for terminals, 66 words for printers).

20/CA Device name:

Passes the logical device number, in ASCII form, of a specific device. The file is assumed to be permanent. If the device name option is specified, the nonshareable device should be ready prior to the HPFOPEN call (otherwise, an error results).

Only one of the following options can be in effect when a file is opened:

itemnum=20

itemnum=22

itemnum=23

itemnum=42

Default: Disk file located on the volume class DISC associated with the group in which file resides.

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example:

%devname% (% is the delimiter, *devname* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

21/@64 Long-mapped:

Returns a long-pointer to the beginning of data of the file. This option maps the file into long pointer space. A long-mapped file can be up to 4GB -64KB or (4,294,901,760 bytes) in size. There is no limit to the number of long-mapped files a process can have open at one time. The pointer can be used as a large array of any type to access the file.

A loaded program file or a loaded library file is not accessible, and a file cannot be loaded with this option.

Sharing long-pointer files is provided through normal file system file

sharing mechanisms. All file system intrinsics applicable to the file can be used. FREAD and FWRITE calls can be mixed with this option.

Standard (STD) disk files of fixed or undefined record length can be accessed with this option. Standard disk files of variable record length can be accessed only if *itemnum*=11 (read-only access). Standard disk files of variable record length and KSAM files can be accessed only if *itemnum*=11 (read-only access) and *itemnum*=17 (set to 1).

Directories may not be opened using long-mapped access.

Default: Not returned

(ASC) Not used for asynchronous devices.

22/CA Volume class:

Passes a character array representing a volume class name where the file space is to be restricted. This option is applicable only at file creation.

A volume class is a subset of volumes within a volume set. The volume class name must be a valid volume class name residing on the volume set bound to the volume (the volume set is an attribute of the group in which the file resides).

Only one of the following options can be in effect when a file is opened with this option:

itemnum=20

itemnum=22

itemnum=23

itemnum=42

Default: A disk file located on the volume class DISC associated with the group in which file resides.

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example:

%volclass% (% is the delimiter, *volclass* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

This *itemnum* may not be specified when creating hierarchical directories.

(ASC) Not used for asynchronous devices.

23/CA Volume name:

Passes a character array representing a volume name that restricts the file specified to a specific volume. The volume must reside within the volume set of the group where the file resides. This option is applicable only at file creation.

Only one of the following options can be in effect when a file is opened with

this option:

itemnum=20

itemnum=22

itemnum=23

itemnum=42

Default: A disk file located on the volume class DISC associated with the group in which the file resides.

This *itemnum* may not be specified when creating hierarchical directories.

(ASC) Not used for asynchronous devices.

24/I32 Density:

Passes the tape density required when writing to a tape file. This option is applicable only when writing to a tape on a drive that supports more than one density. When reading from a tape, the density of the tape overrides this option

Default: The highest density available on the device opened.

(ASC) Not used for asynchronous devices.

25/CA Printer environment:

Passes the name of a file that contains a printer environment. This option is valid only for specified printer devices.

If opening an Hewlett-Packard 268x page printer file, specify an optional printing environment for the job. The printing environment is defined as all of the characteristics of the printed page that are not part of the data itself, including the page size, the margin width, the character set, the orientation (horizontal or vertical), and the name of forms to use. If an environment file is not specified, the file system selects a default printer environment.

Any environment selected remains active until replaced by a new environment, or until a call to FCLOSE (close the printer).

Default: No printer environment file specified

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example,

%envname% (% is the delimiter, *envname* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

26/CA Remote environment:

Passes the node name of the remote computer where the file is located. This option is used when referencing a file located on a remote computer.

Default: No node name passed (local file access)

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example,

%envname% (% is the delimiter, *envname* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

A remote environment cannot be specified when creating or opening files in the HFS name space or in byte stream files.

27/I32 Output priority:

Passes the output priority to be attached to the file for spooled output. This option is applicable only to spooled devices. The output priority must be a number between 1 (lowest priority) and 13 (highest priority), inclusive. If the value specified is less than the current outfence set by the system operator, file printing is deferred until the operator raises the output priority of the file or lowers the outfence. This option can be specified for a file already opened (for example, $STDLIST), where the highest value supplied before the last FCLOSE takes effect. This option is ignored for nonspooled devices.

Default: 8

28/CA Spooled message:

Passes a spooler message associated with a spoolfile. For example, a message is passed that can be used for telling the system operator what type of paper to use in the line printer. This message must be displayed to the system operator and verified before the file can be printed on a line printer. The number of characters allowed <=48; any quantity > 48 characters is truncated.

Default: No spooled message specified

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example:

%message% (% is the delimiter, *message* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

29/I32 Privileged access:

Passes a value that temporarily restricts access to the file number returned from HPFOPEN to a calling process whose execution level is equal to or less than the value specified in this option. This restriction lasts until the file associated with the restricted file number is closed. Do not specify a value less than the execution level of the calling process.

The following values are valid:

0	Privilege level zero (most privileged level)
1	Privilege level one
2	Privilege level two
3	Privilege level three (least privileged level)

Default: The execution level of the calling process

30/I32 Labeled tape type:

Passes a value that indicates tape label type information. This option is valid only for labeled tapes. The following values are valid:

0	ANSI standard labels
1	IBM standard labels

Default: 0

(ASC) Not used for asynchronous devices.

31/CA Labeled tape expiration:

Passes the date of the expiration of the file or the date after which the information in the file is no longer useful, in the format `MM/DD/YY`. The file can be overwritten after this date. If the default is specified, the file can be overwritten immediately. In a volume set, file expiration dates must always be equal to or earlier than the date on the previous file.

Default: No expiration date specified

A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example,

%expdate% (% is the delimiter, *expdate* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

(ASC) Not used for asynchronous devices.

32/CA Labeled tape sequence:

Passes one of the following character arrays indicating the position of the file in relation to other files on the tape:

0	Causes a search of all volumes until the file is found.
`1.. 9999`	Specifies the position of the file relative to the current file on the tape.
`ADDF`	Causes the tape to be positioned so as to add a new file at the end of the volume or last volume in a multivolume set.
`NEXT`	Positions the tape at the next file on the tape. If this is not the first `HPFOPEN/FOPEN` for the file and a rewind occurred on the last close, then the position remains at the beginning of the previous file.

Default: No array passed (no sequence indicated)

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example,

%position% (% is the delimiter, *position* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

(ASC) Not used for asynchronous devices.

33/I32 User labels:

Passes the number, in the range 0..254, of user-label records to be created for the file. Applicable for new disk files only.

Default: 0

This *itemnum* may not be specified when creating hierarchical directories.

(ASC) Not used for asynchronous devices.

34I32 Spooler copies:

Passes a value in the range 1..127 indicating the number of copies of the entire file to be produced by the spooling facility. This option is applicable to spooled devices only. This option can be specified for a file already opened (for example, $STDLIST), where the highest value supplied before the last FCLOSE takes effect. The copies do not appear continuously if the system operator intervenes or if a file of higher output priority becomes READY before the last copy is complete. This option is ignored for nonspooled output devices.

Default: 1

35/I32 File size:

Passes the maximum file capacity:

- For variable-length records, the capacity is expressed in blocks (*blockitem#=recordsize * blockfactor*).
- For fixed-length and undefined-length records, the capacity is expressed in logical records.
- The maximum file size for a standard byte stream file is 2 gigabytes (2,147,483,648 bytes)
- The maximum file size for standard, fixed length record files and KSAM64 files is 128 gigabyte (137,438,953,472 bytes
- The maximum file size for other standard and KSAM XL files is 4,294,901,759 bytes (4 gigabytes less 64K bytes).
- The maximum file size for a CM KSAM data file is 2 gigabytes and for a CM KSAM key file, 2 gigabytes.
- The maximum file size of 500 megabytes, for RIO and circular is

dependent upon both the record size and the number of extents defined for the file:

— For circular and RIO files, *recsize*=256 bytes and *numextent*=32.

This option is applicable only at file creation.

Default: 2 gigabytes

This *itemnum* may not be specified when creating hierarchical directories.

(ASC) Not used for asynchronous devices.

36/I32 Initial allocation:

Passes a positive integer value indicating the number of records to be allocated to the file initially. This option is applicable only at file creation.

Default: 0

This *itemnum* may not be specified when creating hierarchical directories.

(ASC) Not used for asynchronous devices.

37/I32 File code:

Passes a value that can be used as a file code to identify the type of file. This code is recorded in the file label and is accessible through the `FFILEINFO` intrinsic. This option is applicable only at file creation (except when opening an old file that has a negative file code).

If the program is running in user mode, specify a file code in the range 0..32,767 to indicate the file type being created. Programs running in user mode can access files with positive file codes only.

If the program is running in privileged mode, specify a file code in the range -32,768..32,767. Programs running in privileged mode can access files with a file code in the range -32,768..32,767. If an old file is opened that has a negative file code in its file label, the file code specified must match the file code in the file label (otherwise, an error results).

File codes cannot be specified for hierarchical directories. Negative file codes may be used only for files in MPE groups. A *status.info* of `-315` is returned when negative file codes are specified for files outside MPE groups.

Default: 0

(ASC) Not used for asynchronous devices.

38/I32 File privilege:

Passes a value that determines a permanent privilege level to be associated with a newly created file. This option permanently restricts file access to a process whose execution level is less than or equal to the specified value. A value cannot be specified for less than the execution level of the calling process. This option is applicable only at file creation.

The following values are valid:

0	Privilege level zero (most privileged level)
1	Privilege level one
2	Privilege level two
3	Privilege level three (least privileged level)

Default: 3

39/I32 Access type:

Passes a value indicating how to use the file, either sequentially or randomly. The file system uses this information to determine the most efficient prefetching algorithm to improve the performance of the file access.

The following values are valid:

0	Access the file sequentially
1	Access the file randomly

Default: 0

(ASC) Not used for asynchronous devices.

40/I32 Block factor:

Passes the number of logical records to be contained in one physical record (block). This value is used to calculate the physical record size (block size) for disk and magnetic tape files. Valid ranges are 1..32,767. This option is applicable only at file creation.

For fixed-length records, this option specifies the actual number of records in a block. For variable-length records, this option is interpreted as a multiplier used to compute the block size (*record size option * block factor option*). For undefined-length records, this option is always one logical record per block.

This *itemnum* may not be specified when creating hierarchical directories.

Default: 1 for files opened NOBUF; for files opened BUF, it is calculated by dividing the specified records into the block size configured for the device.

(ASC) Not used for asynchronous devices.

41/I32 Name syntax:

Specifies which of three name semantics will be used to interpret the filename passed to HPFOPEN:

0	MPE-escaped semantics
1	MPE-only sematics
2	POSIX semantics

MPE-escaped name semantics is the default value for this *itemnum*. The selected name semantics do not apply to file equations specified through

itemnum 52 or system-defined file references specified through *itemnum* 5.

42/CA Device class:

Passes a device class where the file resides. The file system uses the device class name to select a nonshareable device from a configured list of available devices. The name can have a length of up to eight alphanumeric characters, beginning with a letter (for example, TAPE). If a device class is specified, the file is allocated to any available device in that class.

Only one of the following options can be in effect when a file is opened:

itemnum=20

itemnum=22

itemnum=23

itemnum=42

Default: A disk file located on the volume class DISC associated with the group in which file resides.

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example:

%devclass% (% is the delimiter, *devclass* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

This *itemnum* may not be specified when creating hierarchical directories. Hierarchical directories created on the system volume set is allocated on any volume within the set. Hierarchical directories created on non-system volume sets is allocated on the master volume.

43/record UFID:

Passes a unique file identifier (UFID) to provide a fast opening of an old disk file. A UFID is a record structure, 20 bytes in length, that uniquely identifies a disk file. Using this option avoids a directory search. Obtain the UFID of an opened file by calling FFILEINFO. The UFID can then be passed to HPFOPEN. The file represented by the UFID must be accessible to the process calling HPFOPEN. (All file system security checks are made.) New files cannot be opened with this option. If the file to be opened by the UFID contains a lockword, use *itemnum*=2 to specify the file name with the lockword.

Only files in the MPE name space may be opened by UFID. An attempt to open a file outside the MPE name space by UFID results in a *status.info* of -321 being returned. Only system code may open a file by UFID in the POSIX name space.

Default: No UFID passed (a directory search is performed)

(ASC) Not used for asynchronous devices.

44/I32 Numbuffers:

Passes the number of buffers to allocate to the file. Ignored for standard disk files. This option is useful only for slow devices (such as tapes) used in a buffered mode. Not applicable for files representing interactive terminals; a system-managed buffering method is always used.

The valid range for this option is dependent upon the file type:

- For standard and KSAM files, the valid range is 1..31.
- For circular and RIO files, the valid range is 1..16.
- For message files, the valid range is 2..16. (If a 1 is specified, the file system sets this option to 2 and no error is returned.)

This option must not specify a number of buffers whose combined size exceeds the physical capacity of the file.

This `itemnum` is ignored when creating hierarchical directories.

Default: 2

(ASC) Not used for asynchronous devices.

45/CA Fill character:

Passes two ASCII characters that determine what padding character to use at the end of blocks or unused pages, and the padding used by `itemnum`=53. Do not use delimiter characters for this option. The fill character must be a 2 byte array. The first character only is used as the padding character. The second character is reserved for future use. This option is applicable only at file creation.

Default: Null characters for a binary file and ASCII blanks for an ASCII file.

The default fill character for byte stream files is the null character. The fill character is null for hierarchical directories regardless of the value specified for this `itemnum`.

46/I32 Inhibit buffering:

Passes a value enabling/disabling automatic operataing systembuffering. If `NOBUF` is specified, I/O is allowed to take place directly between the data area and the applicable hardware device.

The following values are valid:

0	Allow normal buffering (`BUF`)
1	Inhibit buffering (`NOBUF`)

Default: 0

`NOBUF` access is oriented to physical block transfer rather than logical record transfer. If `NOBUF` and a variable-length record structure are specified in `itemnum`=6 and the file does not have a variable-length record format, then the format is changed internally to an undefined-length record format. When performing an `FWRITE`, set up the variable structure.

With NOBUF access, responsibility for blocking and deblocking of records in the file belongs to the program. To be consistent with files built using buffered I/O, records should begin on half word boundaries. When the information content of the record is less than the defined record length, you must pad the record with blanks if the file is ASCII, or with zeros if the file is binary.

The record size and block size for files manipulated with NOBUF specified follow the same rules as those files that are created using buffering. The default blocking factor for a file created under NOBUF is one.

If a file is opened NOBUF without multirecord mode specified in *itemnum*=15, then transfer a maximum of only one block of data per read or write.

The end-of-file (EOF) marker, next record pointer, and record transfer count are maintained in terms of logical records for all files. The number of logical records affected by each transfer is determined by the size of the transfer.

Transfers always begin on a block boundary. Those transfers that do not transfer whole blocks leave the next record pointer set to the first record in the next block. The EOF marker always points at the last record in the file.

For files you have opened NOBUF, the FREADDIR, FWRITEDIR, and FPOINT intrinsics treat the *recnum* parameter as a block number.

Indicate non-RIO access to an RIO file by specifying the file NOBUF. Use the physical block size from FFILEINFO to determine the maximum transfer length.

For message files, the file system normally resets *itemnum*=46 to zero. However, a message file can be opened with NOBUF if *itemnum*=17 is set to 1; this determines access to the file record-by-record or by block.

If writing to a message file, open it NOBUF if *itemnum*=17 to access the file block-by-block.

Native byte stream access opens NOBUF (1) regardless of the value specified for this *itemnum*. This *itemnum* is ignored for directories since a physical block transfer interface is not provided for directories.

(ASC) Not used for asynchronous devices.

47/I32 Numextents:

Passes a value in the range 1..32 that determines the number of extents for the file. If a value of 1 is specified, the file is created as one contiguous extent of disk space. If a value greater than 1 is specified, a variable number of extents (with varying extent sizes) is allocated on a need basis. This option is applicable only at file creation.

(ASC) Not used for asynchronous devices.

Default: 1

This *itemnum* may not be specified when creating hierarchical directories.

48/I32 Reverse VT:

Passes a value indicating whether or not the given device name is to be allocated on a remote machine. Specify the remote environment in the same open request, using either the formal designator option or the remote environment option. Reverse VT behaves nearly the same as a terminal opened through remote file access, except that no session is required on the remote machine.

The following values are valid:

0 No reverse VT

1 Reverse VT

Default: 0

49 Reserved for the operating system

50/I32 Final disposition:

Passes a value indicating the final disposition of the file at close time (significant only for files on disk and magnetic tape). A corresponding parameter in a `FILE` command can override this option, unless file equations are disallowed with *itemnum*=9.

The following values are valid:

0 No change. The disposition remains as it was before the file was opened. If the file is new, it is deleted by `FCLOSE`; otherwise, the file is assigned to the domain it belonged to previously. An unlabeled tape file is rewound. If the file resides on a labeled tape, the tape is rewound and unloaded.

1 Permanent file. If the file is a disk file, it is saved in the system file domain. A new or temporary file on disk has an entry created for it in the system (permanent) file directory. If a file of the same name already exists in the directory, an error code is returned at close time and the file remains open. If the file is a permanent file on disk, this domain disposition has no effect. If the file is stored on magnetic tape, the tape is rewound and unloaded.

2 Temporary job file (rewound). The file is retained in your temporary (job/session) file domain and can be requested by any process within your job/session. If the file is a disk file, the uniqueness of the file name is checked. If a file of the same name already exists in the temporary file domain, an error code is returned at close time and the file remains open. When a file resides on unlabeled magnetic tape, the tape is rewound. However, if the file resides on labeled magnetic tape, the tape is backspaced to the beginning of the presently opened file.

3 Temporary job file (not rewound). This value has the same

effect as specifying final disposition option *itemnum*=2, except that tape files are not rewound. In the case of unlabeled magnetic tape, if the FCLOSE is the last done on the device (with no other FOPEN/HPFOPEN calls outstanding), the tape is rewound and unloaded. If the file resides on a labeled magnetic tape, the tape is positioned to the beginning of the next file on the tape.

4 Released file. The file is deleted from the system.

5 Convert a permanent file to a temporary file. The file is removed from the permanent file directory and placed in the temporary file directory. (Privileged mode capability is required to use this option.)

Default: 0

For more information on file disposition at close time, refer to the description of the FCLOSE intrinsic.

(ASC) Not used for asynchronous devices.

51/String Pascal XL string:

Passes a formal file designator that follows MPE/iX file naming conventions, using the Pascal XL STRING type format. This option is identical to *itemnum*=2 except for the type of item. No delimiters are needed.

Default: No string passed

When you use HPFOPEN to open a file, you may use either *itemnum*=2 or you may use *itemnum*=51; you may not use both.

As the default, the formal file designator is interpreted according to MPE-escaped semantics. To choose another syntax, use *itemnum* 41.

The file referred to by *formaldesig* can be either an MPE file (i.e., one that uses MPE syntax) or it can follow HFS syntax. If *formaldesig* follows MPE syntax, the file name can include password, group, and account specifications. The file name can backreference a file equation and optionally be preceded by an asterisk. If *formaldesig* follows HFS syntax, the file name must start with either a dot (.) or a slash (/).

The file referred to by *formaldesig* may reside either in an MPE group or in an HFS directory. For files located in HFS directories, traverse directory entries (TD) access is required to all directories specified in *formaldesig*. If there is no TD access, HPFOPEN fails and a value of -180 is returned in the *status.info* parameter. If you are using HPFOPEN to create a file or hierarchical directory and you lack of create directory entry (CD) access, *status.info* will return a value of -179.

52 File equation string:

Passes a character string that matches the file equation specification syntax exactly. (Refer to the FILE command in the *MPE/iX Commands*

Reference Manual.) This option allows the specification of options available in the FILE command.

The *formaldesig* parameter and *filereference* parameter can contain embedded command interpreter variables and expressions. However, there cannot be more than 8-characters in each of these components (*filename, lockword, groupname, accountname*) including the command interpreter variable and expression characters.

Default: No string passed

A character placed in the first element designates the delimiter used by HPFOPEN to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example:

%fileequation% (% is the delimiter, *fileequation* is the designator)

fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)

The *formaldesignator* in the file equation string must belong to the MPE name space. The *filerefence* in the file equation string is interpreted using MPE-escaped semantics. Both the *formaldesignator* and the *filereference* in the file equation string may also contain embedded command interpreter variables or expressions.

53/I32 ASCII/binary:

Passes a value indicating whether ASCII or binary code is to be used for a new file when it is written to a device that supports both codes. For disk files, this affects padding that can occur when issuing a direct-write intrinsic call (FWRITEDIR) to a record that lies beyond the current logical end-of-file indicator. By default, magnetic tape and files are treated as ASCII files. This option is applicable only at file creation.

The following values are valid:

0	Binary file
1	ASCII file

Default: 0

(ASC) Not used for asynchronous devices.

54/REC KSAM parm:

Passes a record that defines the keys for a new KSAM file.

(KSAM XL and KSAM64) For KSAM XL and KSAM64 files, refer to the *parm* parameter discussion in the *Using KSAM XL*.

(KSAM/3000) The record must be at least 34 bytes in size. For details, refer to the *ksamparam* parameter discussion in the *KSAM/3000 Reference Manual*.

Default: No record passed

	(ASC) Not used for asynchronous devices.
55	Reserved for the operating system
56/I32	Object class:
	Passes a user object class number, in the range 0..10, that is associated with the file.
	Default: Determined by the file code for system and subsystem files, and by the file type and record type for normal user files.
57	Reserved for the operating system
58	Reserved for the operating system
59	Reserved for the operating system
60	Reserved for the operating system
61	Reserved for the operating system
64/BA	Access Control Definition:

Passes a byte array defining the access control definition (ACD) to be attached to a new file. The byte array has a length of 1 to 279 bytes. Unlike other `HPFOPEN` options that expect a delimiter at both the beginning and the end of the byte array, this option only expects a trailing carriage return character as a delimiter, for example,

```
(X:@.@;R,W:MGR.SYS;RACD:JOHN.SMITH)<cr>
```

Where, the <cr> is the carriage return character (`13`, `0x0D`).

The ACD assigned to a newly created file or directory may be different from the ACD specified as the value of this *`itemnum`*. If a process' file mode creation mask (`cmask`) is initialized, it modifies the ACD.

65-76	Reserved for the operating system
77/I32	Data format

Allows the caller to select a different format to view the data in the file. The current valid values for this item are:

0	Use the standard record based view of accessing the file. This is the default value for all opens. For conventional files, the file is record based. For directories, the standard non-privileged directory information is returned when the `HPDIRREAD` intrinsic is called. When this value is specified with a byte stream file, access to the file is emulated to appear like a buffered variable record file. This is the default.
1	When this value is specified, calls to `HPDIRREAD` returns privileged directory information including the UFID and link ID of the entry. Note that this value is only applicable to directory files. This value is ignored for all other file types. In order to specify this value, the caller must be

executing in system code.

2 — When this value is specified, the system attempts to let the caller access the file as a native byte stream file. Byte stream emulation is supported for ordinary fixed and variable length record files as well as for files with the byte stream record type. If this value is requested against a file type that does not support byte stream access, an error is returned.

Specifying any value other than the values described above will result in an error.

79/I32 POSIX Non-Block Mode

Specifying this itemnum allows the caller to open a file with the POSIX Non-Block mode. This item is useful for a subset of files (including pipes and FIFO's) and is ignored for all other files. The behavior of the HPFOPEN call with this option is dependent on the type of file being opened.

The current valid values for this item are:

0 — This value indicates that Non-Block mode is off. This is the default value.

1 — This value indicates that Non-Block mode is on.

Specifying any value other than those described above will result in error.8

80/I32 Reserved for the operating system.

81/I32 Symbolic link option:

This *itemnum* allows the caller to specify different options when traversing through or opening a symbolic link. The valid values for this *itemnum* are described below:

0 — Follow symbolic links. This is the default value for this option. When a symbolic link is encountered it is traversed according to the path specified in the symbolic link.

1 — Does not follow symbolic links. If the final component of a path is a symbolic link, then no traversal is done and the symbolic link is opened. Symbolic links that occur prior to the last component is traversed.

Specifying any value other than those above will result in error.

82-86 Reserved for MPE/iX

87/@64 Large Mapped Access

Returns a pointer to the beginning of file data. This option can be used on any sized file, but is the only means by which to open files larger than 4 gb -64kb (4,294,901,760 bytes) for mapped access. Large mapped access shares the same constraints on file types as the long mapped option (option 21).

Optional parameters you can use to specify file access and device control characteristics are described elsewhere in this manual. For more details on the HPFOPEN intrinsic, refer to the *MPE/iX Intrinsics Reference Manual*.

The FOPEN Intrinsic

The `FOPEN` intrinsic is the other programmatic interface for supplying the file system with information about your file. Its syntax is:

```
filenum := FOPEN  (formaldesignator, foptions, aoptions,
                   recsize, device, formmsg, userlabels,
                   blockfactor, numbuffers, filesize,
                   numextents, initialloc, filecode);
```

The following table shows the correspondence between the optional parameters of `FOPEN` and `HPFOPEN` that you can use to specify a file's physical characteristic at file creation. For more details on using the `FOPEN` intrinsic, refer to the *MPE/iX Intrinsics Reference Manual*.

Table 2-1. FOPEN/HPFOPEN Parameter Equivalents

FOPEN Parameter	HPFOPEN Itemnum,Item
filenum (functional return)	*filenum* (parameter)
formaldesig	*2, formaldesig*
foption: Bits (14:2) domain Bit (13:1) ASCII/binary Bits (10:3) file designator Bits (8:2) record format Bit (7:1) carriage-control Bit (6:1) labeled tape Bit (5:1) disallow file equation Bits (2:3) file type	 *3, domain* *53, ASCII/binary* *5, file designator* *6, record format* *7, carriage-control* *8, labeled tape* *9, disallow file equation* *10, file type*
aoption: Bits (12:4) access type Bit (11:1) multirecord Bit (10:1) dynamic locking Bits (8:2) exclusive Bit (7:1) inhibit buffering Bits (5:2) multiaccess mode Bit (4:1) nowait I/O Bit (3:1) file copy	 *11, access type* *15, multirecord* *12, dynamic locking* *13, exclusive* *46, inhibit buffering* *14, multiaccess mode* *16, nowait I/O* *17, file copy*
recsize	*19, record size*

Table 2-1. FOPEN/HPFOPEN Parameter Equivalents

FOPEN Parameter	HPFOPEN Itemnum,Item
device	*20, device name* *22, volume class* *23, volume name* *24, density* *25, printer environment* *26, remote environment* *42, device class* *48, reverse VT*
formmsg	*8, labeled tape label* *28, spooled message* *30, labeled tape type* *31, labeled tape expiration* *32, labeled tape sequence* *54, KSAM parms*
userlabels	*33, user labels*
blockfactor	*40, block factor*
numbuffers: Bits (11:5) numbuffers Bits (4:7) spooler copies Bits (0:4) output priority	*44, numbuffers* *34, spooler copies* *27, output priority*
filesize	*35, filesize*
numextent	*47, numextent*
initialloc	*36, initial allocation*
filecode	*37, filecode*

Table 2-2. Determining a File's Physical Characteristics Using FOPEN

Characteristic	Parameter Description	Default Value
Record Structure	*foptions*: ASCII/binary option *foptions*: record format option *foptions*: carriage-control option *recsize* parameter	Binary Fixed-length None 256 bytes

Table 2-2. Determining a File's Physical Characteristics Using FOPEN

Characteristic	Parameter Description	Default Value
File Structure	*foptions*: file type option *device* parameter *blockfactor* parameter *filesize* parameter *initialloc* parameter *numextents* parameter	Standard Volume class DISC 128/*recsize* rounded down 4 gigabytes 0 extents A variable number of extents is allocated
File Identification	*userlabels* parameter *filecode* parameter	No user labels *filecode* = 0

The BUILD Command

The BUILD command creates a file in much the same way as the HPFOPEN/FOPEN intrinsic, except that HPFOPEN/FOPEN is used within a program and BUILD is entered as an MPE/iX command.

The parameters for BUILD have meanings and applications that are similar to the corresponding parameters for HPFOPEN/FOPEN. For more information about how to use the BUILD command, refer to the *MPE/iX Commands Reference Manual*.

The FILE Command

The FILE command is used to determine how a file will be accessed. You may use FILE to describe any of the characteristics available with HPFOPEN/FOPEN or BUILD, but you cannot actually create a file with the FILE command. While HPFOPEN/FOPEN and BUILD physically allocates space for a file and define its characteristics, the FILE command may only define how a file will be accessed at run time.

To be effective, a FILE command must be issued before your file is opened; it takes effect when the file is opened. A FILE command remains in effect until the job or session ends, until it is canceled with a RESET command, or until it is overridden by another command for the same file. Thus, if you enter a FILE command equating the formal designator DATAFL to the actual designator DISCFILE (indicating a disk file) and then run three programs that reference DATAFL, all three programs will access the file DISCFILE. If you wish to define other characteristics for the file, simply issue another FILE command; if you want to nullify the FILE command completely so that the formal designator has the characteristics originally specified by the program that is using it, issue a RESET command.

For example, suppose that you run two programs, both referencing a new temporary file named DFILE located on disk. Before you un the first program, you want to redefine the file so that it is output to the standard list device. To do this, you would issue a FILE command equating DFILE with the actual designator $STDLIST. In the second program, the file is again to be a temporary file on disk. You issue a RESET command so that the specifications supplied by the second program (rather than those in the FILE command) apply.

```
 JOB JNAME,UNAME.ANAME
.
.
.
 FILE DFILE=$STDLIST
 RUN PROG1
.
.
.
 RESET DFILE
 RUN PROG2
.
.
.
```

A comparison of the parameters for FILE, FOPEN, and HPFOPEN is given in Table 2-3. on page 52 For more information about using the FILE command, refer to the *MPE/iX Commands Reference Manual*.

Table 2-3. FILE, FOPEN, and HPFOPEN Parameters

CHARACTERISTIC	:FILE\ PARAMETER	FOPEN\ PARAMETER	HPFOPEN\ PARAMETER
Formal file designator	*formal designator*	*formaldesignator*	*formaldesignator option* (*itemnum=2*)
Actual file designator	*filereference* $NEWPASS $OLDPASS $NULL $STDIN $STDINX $STDLIST	Default file designator *foption* (Bits 10:3)	*designator option* (*itemnum=5*)
Domain	NEW OLD OLDTEMP	Domain *foption* (Bits 14:2)	*domain option* (*itemnum=3*)
Logical record size	*recsize*	*recsize*	*record size option* (*itemnum=19*)
Block/buffer size	*blockfactor*	*blockfactor*	*block factor option* (*itemnum=40*)
Record format	F V U	Record format *foption* (Bits 8:2)	*record format option* (*itemnum=6*)
ASCII/Binary Code	ASCII Binary	ASCII/Binary *foption* (Bits 13:1)	*ASCII/Binary option* (*itemnum=53*)
Carriage-control characters supplied in FWRITE	CCTL NOCCTL	Carriage-control *foption* (Bits 7:1)	*carriage-control option* (*itemnum=7*)
Access mode	IN OUT OUTKEEP APPEND INOUT UPDATE	Access-type *aoption* (Bits 12:4)	*access type option* (*itemnum=11*)
Number of buffers	*numbuffers* NOBUF	*numbuffers* (Bits 11:5)	*numbuffers option* (*itemnum=44*)
Exclusive/Share access	EXC SEMI SHR	EXCLUSIVE access *aoption* (Bits 8:2)	*exclusive option* (*itemnum=13*)

Table 2-3. FILE, FOPEN, and HPFOPEN Parameters

CHARACTERISTIC	:FILE\ PARAMETER	FOPEN\ PARAMETER	HPFOPEN\ PARAMETER
Multi access	MULTI NOMULTI GMULTI	Multiaccess mode *aoption* (Bits 5:2)	*multiaccess option* (*itemnum=14*)
Multirecord	MR NOMR	Multirecord *aoption* (Bits 11:1)	*multirecord option* (*itemnum=15*)
File disposition	DEL SAVE TEMP	N/A	*final disposition option* (*itemnum=50*)
Device class name or logical device number	*device*	*device*	*device class option* (*itemnum=42*) *device name option* (*itemnum=20*)
Output priority	*outputpriorit y*	*numbuffers* (Bits 0:4)	*output priority option* (*itemnum=27*)
NOWAIT input/output	NOWAIT WAIT	NOWAIT I/O *aoption* (Bits 4:1)	*nowait I/O option* (*itemnum=16*)
Number of copies	*numcopies*	*numbuffers* (Bits 4:7)	*spooler copies option* (*itemnum=34*)
File code	*filecode*	*filecode*	*filecode option* (*itemnum=37*)
File capacity	*numrec*	*filesize*	*filesize option* (*itemnum=35*)
Total number of extents	*numextents*	*numextents*	*numextents option* (*itemnum=47*)
Extents initially allocated	*initalloc*	*initalloc*	*initial allocation option* (*itemnum=36*)
FILE command prohibition	N/A	Disallow FILE equation *foption* (Bits 5:1)	*disallow file equation option* (*itemnum=9*)
Dynamic file locking	LOCK NOLOCK	*Dynamic locking aoption* (Bits 10:1)	*disallow file equation dynamic locking option* (*itemnum=12*)
Forms-alignment message	FORMS	*formmsg*	*spooled message option* (*itemnum=28*)

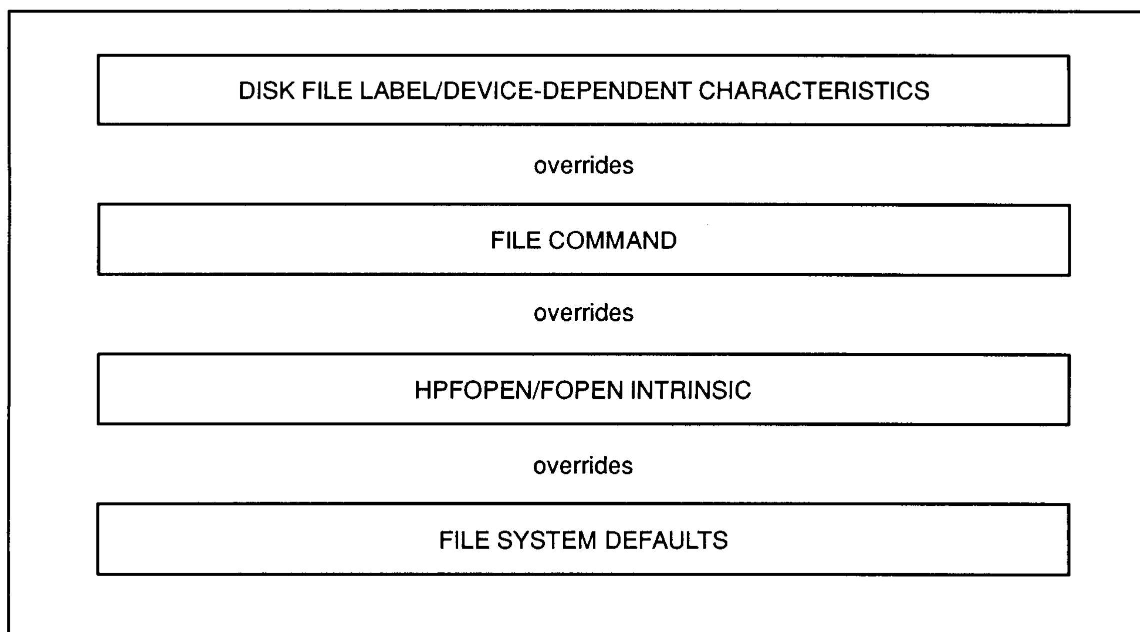
Table 2-3. FILE, FOPEN, and HPFOPEN Parameters

CHARACTERISTIC	:FILE\ PARAMETER	FOPEN\ PARAMETER	HPFOPEN\ PARAMETER
User labels for disk file	N/A	*userlabels*	*user labels option* (*itemnum=33*)
File labels for magnetic tape files	LABEL NOLABEL	Labeled tape *foption* (Bit 6:1)	*labeled tape label option* (*itemnum=8*)
File type	STD CIR MSG RIO	file type *foption* (Bits 2:3)	*file type option* (*itemnum=10*)
Mapped access method	N/A	N/A	*short mapped option* (*itemnum=18*) *long mapped option* (*itemnum=21*) *large mapped option (itmenum=87)*
Restrict file access according to execution level	N/A	N/A	*privileged access option* (*itemnum=29*) *file privilege option* (*itemnum=38*)
Determine optimum pre-fetch algorithm	N/A	N/A	*will access option* (*itemnum=39*)
Fast file open	N/A	N/A	*UFID option* (*itemnum=43*)
Fill character for record padding	N/A	N/A	*fill character option* (*itemnum=45*)
Formal file designator Pascal/iX string type	N/A	N/A	*Pascal/iX string option* (*itemnum=51*)
File equation string for file open	N/A	N/A	*file equation string option* (*itemnum=52*)
KSAM key file record	N/A	KSAM param	*KSAM parm option* (*itemnum=54*)
User object class number	N/A	N/A	*object class option* (*itemnum=56*)

Summary of Overrides

If a FILE command has been entered that contradicts some of the HPFOPEN/FOPEN parameters for a file, which takes precedence? What happens if some parameters are left out? The file system maintains a hierarchy of overrides for just such situations (illustrated in Figure 2-1.):

Figure 2-1. File System Hierarchy of Overrides



Since the physical characteristics of a file cannot be changed after it has been created, it makes sense that the file label would take precedence over all commands. Other determinants are effective only when a new file is being created.

NOTE FILE commands and HPFOPEN/FOPEN calls cannot alter physical characteristics of an existing file.

Specifying a Record Format

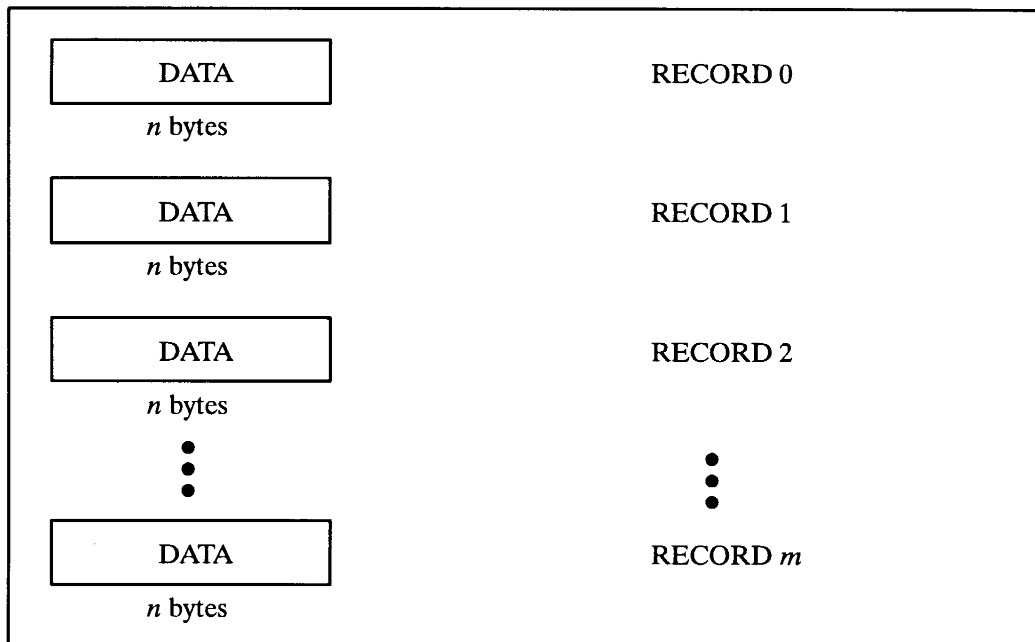
A file can contain records written in only one of three formats: fixed-length, variable-length, and undefined-length. You can specify the format that you want for your records, either with the `HPFOPEN/FOPEN` intrinsic or the MPE/iX `BUILD` or `FILE` commands. Files residing on disk or magnetic tape may contain records in any of the three formats. For files on other devices, the file system overrides any specifications that you supply, and treats the records as undefined-length records.

Fixed-length records

When you create a file and request fixed-length records, all the records in the file will be the same size. The file system knows how much space has been allocated for each record, and that all of the space is to be available for data.

Figure 2-2. depicts a file with fixed-length records. A record size of *n* bytes has been specified. Note that each record is the same size and contains the same amount of information.

Figure 2-2. Fixed-Length Records



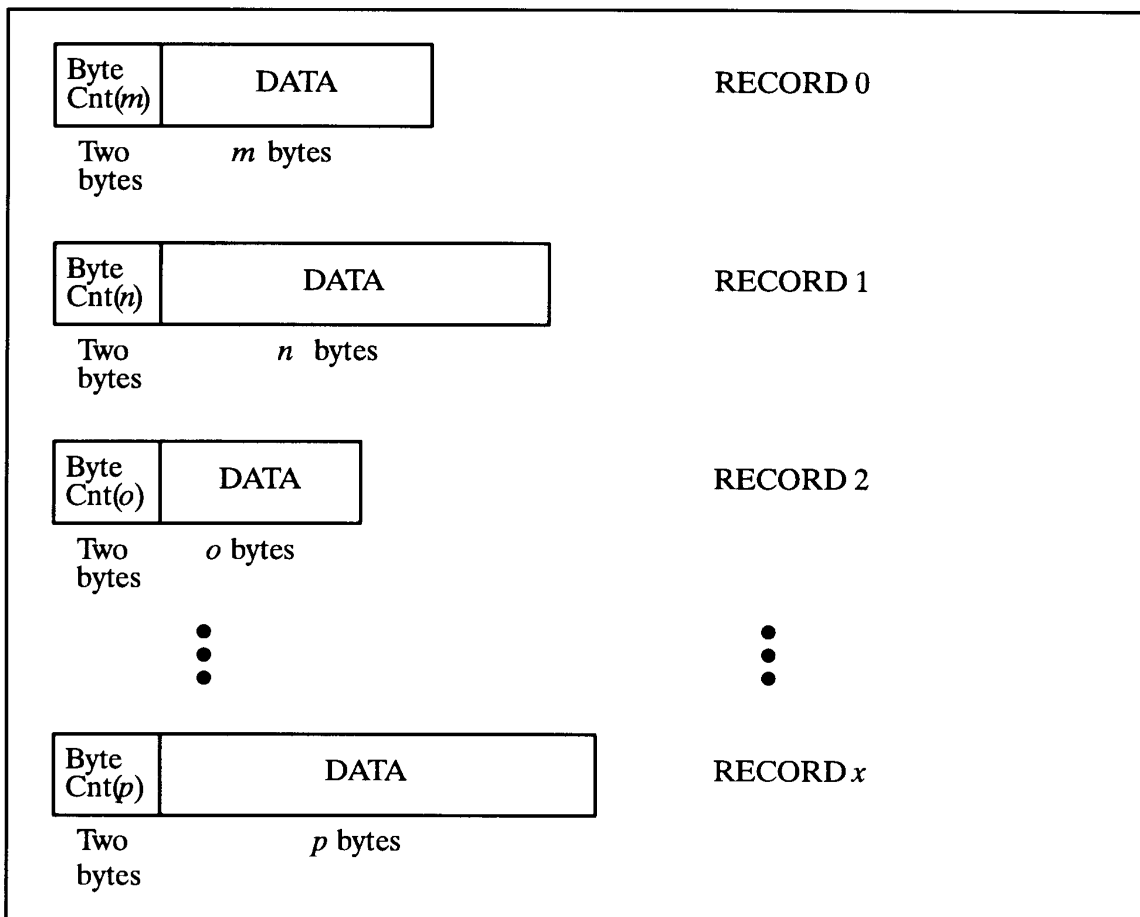
Variable-length records

There may be a time when you want a disk file in which the logical records need not be the same size. In this case, you can request that the format of the records be variable-length. The file system knows the size of each logical record because each record is preceded by a two-byte (16-bit) counter giving the length of the record in bytes; thus, the data for each

record is accompanied by an indication of its length. When you build a file containing variable-length records, specify a record size at least large enough to accommodate your longest record.

Figure 2-3. depicts a file with variable-length records. The byte count preceding the first byte of each record gives its record's length.

Figure 2-3. Variable-length Records



Undefined-length records

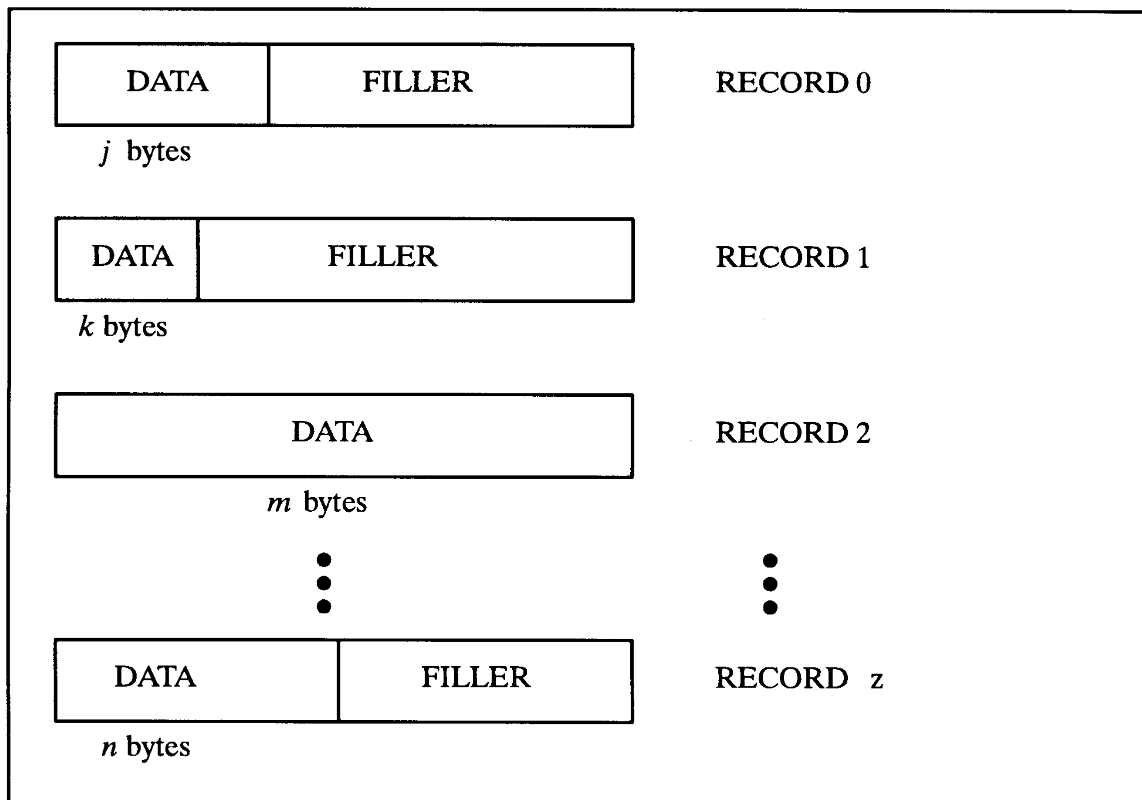
When your file contains undefined-length records, the file system does not know the amount of good data in any given logical record. The data length is "undefined." Undefined-length records are especially useful when you are reading tapes of unknown record length produced on other systems.

The file system knows the maximum room available in each record because the same amount of space is allocated for each record; however, the data in the records may vary in length, so MPE/iX pads the unused space with "filler" instead of good data. The file system supplies this filler during writes to the file when the length of the data being written is less than the maximum record length. The file system cannot distinguish between valid data and filler. When you read data from a file you must be able to distinguish between the valid

data and the filler.

Figure 2-4. depicts a file with undefined-length records. When data does not fill the space allocated, filler occupies the unused space.

Figure 2-4. Undefined-Length Records



The three record formats, fixed-length, variable-length, and undefined-length are summarized in Table 2-4. on page 58

Table 2-4. Comparison of Logical Record Formats

Fixed-Length	Variable-Length	Undefined-Length
Data length known to file system.	Data length known to file system.	Data length not known to file system.
Same length for all records.	Record length varies.	Same length for all records.
Record space contains data only.	Record space contains data plus byte count.	Record space contains data plus filler.
Request actual size for records.	Request maximum size for records.	Request maximum size for records.

Specifying a File Type

When you create, a file the file system imposes a structure and access method on the contents of the file. The file system allows you to access the records in a file only in the manner dictated by the file type that you specified at file creation. Depending upon your intended use of the file, you can specify four special file types in addition to the standard file type: KSAM files, RIO files, circular files, and message files.

Standard files

By far the most common type of file is the standard file, a structure comprised simply of a group of records beginning with record 0 and ending with record *n - 1* (where *n* is the maximum specified in the *filesize option*). Examples of standard files are Editor files and program files. A standard file is the default file type created when you first open a file.

KSAM files

The keyed sequential access method (KSAM) is a method of organizing records in a file according to the content of key fields within each record. Every record in a KSAM file contains a primary key field whose contents determine the primary logical sequence of records in the file. Other key fields can also be defined so that the file can be sequenced in alternate orders. The order in that the records are physically written to the file, the chronological order, can be the same as the primary key sequence or it can be unrelated to any logical sequence.

KSAM files are not dealt with in this manual. Instead, the creation and application of KSAM files are discussed in great detail in the *KSAM/3000 Reference Manual* and *Using KSAM/XL*.

RIO files

RIO is a random access method that permits individual file records to be deactivated. These inactive records retain their relative position within the file. RIO files are intended for use primarily by COBOL programs; however, you can access these files by programs written in any language.

RIO files may be accessed in two ways: RIO access and non-RIO access. RIO access ignores the inactive records when the file is read sequentially using the FREAD intrinsic, and these records are transparent to you; however, they can be read by random access using FREADDIR. They may be overwritten both serially and randomly using FWRITE, FWRITEDIR, or FUPDATE. With RIO access, the internal structure of RIO blocks is transparent.

Circular files

Circular files are wrap-around structures that behave as standard sequential files until they are full. As records are written to a circular file, they are appended to the tail of the file; when the file is filled, the next record added causes the block at the head of the file to be deleted and all other blocks to be logically shifted toward the head of the file. Circular

files may not be simultaneously accessed by both readers and writers. When the file has been closed by all writers, it may be read; a reader takes records from the circular file one at a time, starting at the head of the file.

Circular files are particularly useful as history files, when a user is interested in the information recently written to the file and is less concerned about earlier material that has been deleted. These history files are frequently used as debugging tools. Diagnostic information may be written to the file, and the most recent and relevant material can be saved and studied.

Creating a circular file is similar to creating a message file. When a user process opens a new file and indicates that it will be a circular file, the HPFOPEN/FOPEN intrinsic creates the new circular file. In order to create a circular file with the BUILD command, use the CIR keyword; for example, to build a circular file named CIRCLE, enter:

```
BUILD CIRCLE;CIR
```

A new circular file may also be specified with a FILE command. Use the CIR keyword for a new file:

```
FILE ROUND, NEW; CIR
```

A circular file named ROUND is indicated.

When you perform a LISTFILE,2 command, circular files are identified by an "O" in the TYP field; CIRCLE is identified here:

```
FILENAME CODE -----------LOGICAL RECORD----------- ----SPACE----

                       SIZE TYP EOF LIMIT R/B      SECTORS #X MX

CIRCLE                 256B FBO   0  1023   1            0   0  8
```

Message files

Message files are used by interprocess communication (IPC), a facility of the file system that permits multiple-user processes to communicate with one another in an easy and efficient manner. Message files act as first-in-first-out queues of records, with an entry made by FWRITE and a deletion made by FREAD; one process may submit records to the file with the FWRITE intrinsic while another process takes records from the file using the FREAD intrinsic.

Message files are not dealt with in this manual. Instead, the creation and application of message files are discussed in great detail in the *IPC Communication Programmer's Guide*.

Specifying Record Size

You can specify the size of the records in your file by using the `BUILD` (for disk files) or `FILE` commands, or the `HPFOPEN/FOPEN` intrinsic; however, the interpretation of the requested record size can be affected by the record structure and data format chosen as well as the device for the file.

NOTE Within MPE/iX and in various subsystems, the record size for an ASCII file is usually identified in terms of bytes (8 bits) and the record size for a binary file is identified in terms of half-words (16 bits). This convention is a matter of convenience only, since most users think of ASCII files as being character oriented.

When you specify the record size for a fixed-length ASCII file, the record size determined for the file is the same as that which you specified for it. The maximum record size allowed for fixed-length ASCII files is 32767 bytes.

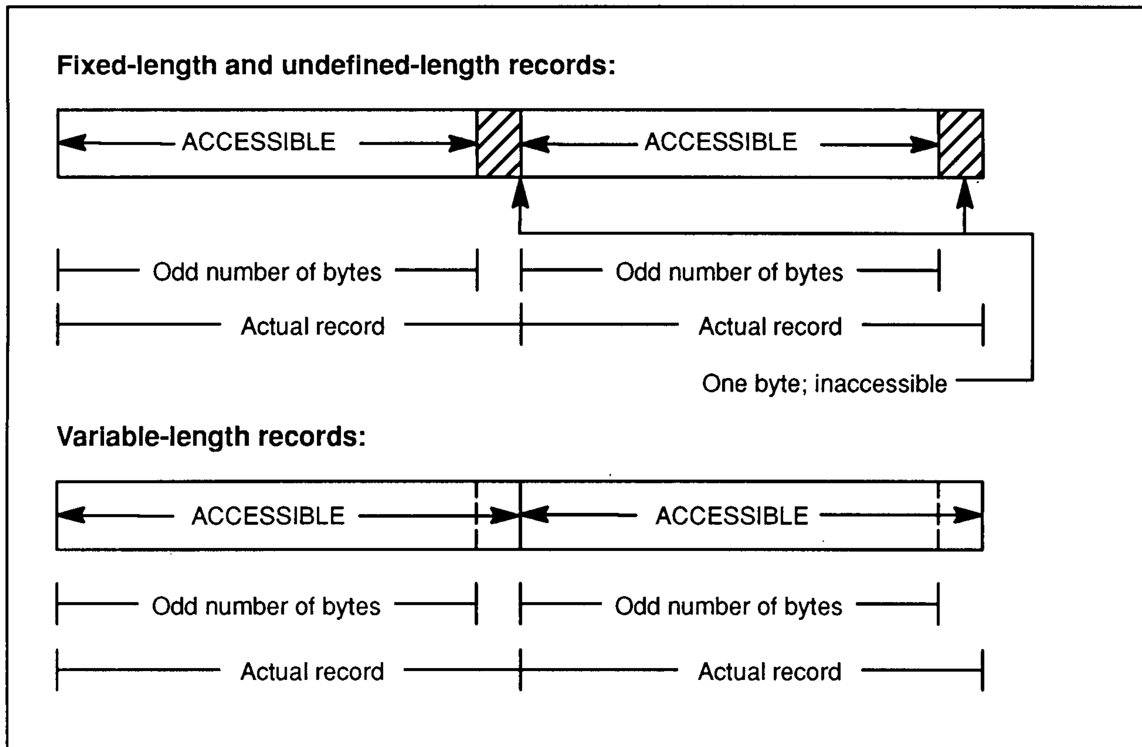
To maintain compatibility with pre-900 Series HP 3000 computer systems, the following file types always begin records on half-word boundaries:

- ASCII files with variable-length or undefined-length record format
- binary files with fixed-length, variable-length, or undefined-length record format

For these file types, when you specify an odd-byte record size, the file system rounds up the value to an even number to reflect the fact that these records always begin on half-word boundaries. The maximum record size you can specify for these file types is 32766 bytes.

When the file is a binary file or a variable-length ASCII file, the extra byte is available to be used for data. Figure 2-5. illustrates the placement of odd-bytes records and the disposition of the added byte.

Figure 2-5. Record Placement for ASCII Files



Rather than specify your own record size, you can accept the default record size for the device that you are using. Default record sizes are listed in Table 2-5. on page 62 Note that subsystem defaults may be different from MPE defaults; for example, the Editor default may be 72 or 80 bytes (depending on text format) while the MPE standard default is the record size configured for the device.

Table 2-5. Standard Default Record Sizes

DEVICE	RECORD SIZE (BYTES)
Disk	256
Magnetic Tape Unit	256
Terminals (most cases)	80
Line Printer	132
Plotter	510
Programmable Controller	256
Synchronous Single-Line Controller	256

Specifying Disk Volume Restrictions

MPE/iX makes a distinction between the device and the media. The device is the disk drive and the media is the disk pack. The MPE/iX volume management facility controls the media and divides the media into three entities:

- Volume set, a set of related disk packs assigned to the group in which you create your file.
- Volume class, a subset of a volume set. A volume can be assigned to more than one volume class.
- Volume, a single disk pack. Each volume on a system is a member of a volume set.

By default, when you create (or otherwise access) a disk file and data is posted to disk, new extents are placed wherever space is available on any volume within the volume class `DISC` assigned to the group in which your file resides.

You can use either the *`volume name option`* or the *`volume class option`* of `HPFOPEN`, or the *`device`* parameter of `FOPEN`, to specify either a volume name or a volume class name, thus restricting the placement of your file's extents to either the specified volume or to the specified volume class within the volume set.

The *`device`* parameter of `FOPEN` also allows you to specify a volume name or a volume class in an additional manner, due to the necessity of maintaining `FOPEN` compatibility with MPE V/E based computer systems. If an LDEV (a logical device number used to identify a device) is passed into `FOPEN`, MPE/iX translates the LDEV into the volume name that is currently mounted on the disk device and places the volume name in the file's label. Similarly, a device class that is associated to a disk device is translated into the volume class name.

NOTE `HPFOPEN` fails if, when creating a disk file, an LDEV is passed to *`volume name option`* for a mounted disk or a device class is passed to *`volume class option`*.

Specifying a File Code

MPE/iX subsystems often create special-purpose files whose functions are identified by four-digit integers called file codes, written in their system file labels. HPFOPEN/FOPEN, BUILD, and FILE have parameters that enable you to specify a file code for your file when you first create it. For user files, you can use as a file code any number from 0 through 1023. Numbers above 1023 listed in Table 2-6. on page 64 are predefined by Hewlett-Packard for special system files and should not be redefined for your use. If you do not specify a file code when you create a file, the MPE/iX default value of zero applies.

For instance, compilers create Native Mode Object (NMOBJ) files, written in a special format and identified by the code 1461, upon which they compile object programs. User programs sometimes create files that must be identified in some unique way, too. Such a program might produce a permanent disk file identified by the integer 1. If you were to run this program several times and want to uniquely identify the file produced on each run (or set of runs) by a special class, purpose, or function, you could use a FILE command to supply a unique file code for each run (or group of runs).

For instance, on the second run, you might wish to classify the file with the file code 2, as follows:

```
                          File code
                          |
FILE DESGX=DESGB;CODE=2
RUN FILEPROD
```

If you later wished to determine the classification to which this file belonged, you could use the LISTFILE command with an information level of 1, which prints the file name, file code, and other information about the file. Alternatively, you could determine the file code by calling the FFILEINFO intrinsic. Both LISTFILE and FFILEINFO are discussed in *Getting System Information Programmer's Guide*. The file codes that have particular Hewlett-Packard-defined meanings are listed in Table 2-6. on page 64

Table 2-6. Reserved File Codes

Integer	Mnemonic	Meaning
1024	USL	User Subprogram Library
1025	BASD	Basic Data
1026	BASP	Basic Program
1027	BASFP	Basic Fast Program
1028	RL	Compatibility Mode Relocatable Library
1029	PROG	Compatibility Mode Program File
1031	SL	Segmented Library
1035	VFORM	VPLUS Forms File

Table 2-6. Reserved File Codes

Integer	Mnemonic	Meaning
1036	VFAST	VPLUS Fast Forms File
1037	VREF	VPLUS Reformat File
1040	XLSAV	Cross Loader ASCII File (SAVE)
1041	XLBIN	Cross Loader Relocated Binary File
1042	XLDSP	Cross Loader ASCII File (DISPLAY)
1050	EDITQ	Edit Quick File
1051	EDTCQ	Edit KEEPQ File (COBOL)
1052	EDTCT	Edit TEXT File (COBOL)
1054	TDPDT	TDP Diary File
1055	TDPQM	TDP Proof Marked QMARKED
1056	TDPP	TDP Proof Marked Non-COBOL File
1057	TDPCP	TDP Proof Marked COBOL File
1058	TDPQ	TDP Work File
1059	TDPXQ	TDP Work File (COBOL)
1060	RJEPN	RJE Punch File
1070	QPROC	QUERY Procedure File
1080	KSAMK	KSAM Key File
1083	GRAPH	GRAPH Specification File
1084	SD	Self-Describing File
1090	LOG	User Logging Log File
1100	WDOC	HP WORD Document
1101	WDICT	HP WORD Hyphenation Dictionary
1102	WCONF	HP WORD Configuration File
1103	W2601	HP WORD Attended Printer Environment
1110	PCELL	IFS/3000 Character Cell File
1112	PENV	IFS/3000 Environment File
1113	PCCMP	IFS/3000 Compiled Character Cell File
1114	RASTR	Graphics Image in RASTER Format
1130	OPTLF	OPT/3000 Log File

Table 2-6. Reserved File Codes

Integer	Mnemonic	Meaning
1131	TEPES	TEPE/3000 Script File
1132	TEPEL	TEPE/3000 Log File
1133	SAMPL	APS/3000 Log File
1139	MPEDL	MPEDCP/DRP Log File
1140	TSR	HPToolset Root File
1141	TSD	HPToolset Data File
1145	DRAW	Drawing File for HPDRAW
1146	FIG	Figure File for HPDRAW
1147	FONT	Reserved
1148	COLOR	Reserved
1149	D48	Reserved
1152	SLATE	Compressed SLATE File
1153	SLATW	Expanded SLATE Work File
1156	DSTOR	RAPID/3000 DICTDBU Utility Store File
1157	TCODE	Code File for Transact/3000 Compiler
1158	RCODE	Code File for Report/3000 Compiler
1159	ICODE	Code File for Inform/3000 Compiler
1166	MDIST	HP Desk Distribution List
1167	MTEXT	HP Desk Text
1168	MARPA	ARPA Messages File
1169	MARPD	ARPA Distribution List
1170	MCMND	HP Desk Abbreviated Commands File
1171	MFRTM	HP Desk Diary Free Time List
1172	None	Reserved
1173	MEFT	HP Desk External File Transfer Messages File
1174	MCRPT	HP Desk Encrypted Item
1175	MSERL	HP Desk Serialized (Composite) Item
1176	VCSF	Version Control System File
1177	TTYPE	Terminal Type File

Table 2-6. Reserved File Codes

Integer	Mnemonic	Meaning
1178	TVFC	Terminal Vertical Format Control File
1192	NCONF	Network Configuration File
1193	NTRAC	Network Trace File
1194	NTLOG	Network Log File
1195	MIDAS	Reserved
1211	NDIR	Reserved
1212	INODE	Reserved
1213	INVRT	Reserved
1214	EXCEP	Reserved
1215	TAXON	Reserved
1216	QUERF	Reserved
1217	DOCDR	Reserved
1226	VC	VC File
1227	DIF	DIF File
1228	LANGD	Language Definition File
1229	CHARD	Character Set Definition File
1230	MGCAT	Formatted Application Message Catalog
1236	BMAP	Base Map Specification File
1242	BDATA	HP Business BASIC/V Data File
1243	BFORM	HP Business BASIC/V Field Order File for VPLUS
1244	BSAVE	HP Business BASIC/V SAVE Program File
1245	BCNFG	Configuration File for Default Options for HP Business BASIC Programs
1246	BKEY	Function Key Definition File for Terminal
1258	PFSTA	Pathflow STATIC File
1259	PFDYN	Pathflow Dynamic File
1270	RFDCA	Revisible Form DCA Data Stream
1271	FFDCA	Final Form DCA Data Stream
1272	DIU	Document Interchange Unit File
1273	PDOC	HP WORD/150 Document

Table 2-6. Reserved File Codes

Integer	Mnemonic	Meaning
1275	DFI	DISOSS Filing Information File
1276	SRI	Search Restart Information File
1401	CWPTX	Chinese Word Processor Text File
1421	MAP	HP MAP/3000 Map Specification File
1422	GAL	Reserved
1425	TTX	Reserved
1428	RDIL	HP Business Report Writer (BRW) Dictionary File CM
1429	RSPEC	BRW Specification File
1430	RSPCF	BRW Specification File
1431	REXCL	BRW Execution File
1432	RJOB	BRW Report 509 File
1433	ROUT1	BRW Intermediate Report File
1434	ROUTD	BRW Dictionary Output
1435	PRINT	BRW Print File
1436	RCONF	BRW Configuration File
1437	RDICN	BRW NM Dictionary File
1438	REXNUM	BRW NM Execution File
1441	PIF	Reserved
1476	TIFF	Tag Image File Format
1477	RDF	Revisible Document Format
1478	SOF	Serial Object File
1479	GPF	Chart File for Charting Gallery Chart
1480	GPD	Data File for Charting Gallery Chart
1483	VCGPM	Virtuoso Core Generator Processed Macro File
1484	FRMAT	Formatter
1485	DUMP	Dump Files Created and Used by IDAT and DPAN
1486	NNMD0	New Wave Mail Distribution List
1491	X4HDR	X.400 Header for HP Desk Manager
1500	WP1	Reserved

Table 2-6. Reserved File Codes

Integer	Mnemonic	Meaning
1501	WP2	Reserved
1502	LO123	Lotus 123 Spread Sheet
1514	FPCF	Form Tester Command Spec File
1515	INSP	Spooler/XL Input Spool File
1516	OUTSP	Spooler/XL Output Spool File
1517	CHKSP	Spooler/XL Checkpoint Spool File
1521	DSKIT	HP Desk Intrinsics Transaction File
1526	MSACK	Man Server Acknowledgement
1527	MSNON	Man Server Non-Delivery Notification
1528	MSTRC	Man Server Trace File
3333		Reserved

NOTE Default is the unreserved file code of 0.

Using `1090 (LOG)` as a designated file code may not yield the number of records that you specify in the `DISC=` parameter. Most files use the number of records specified in the `DISC=` parameter as the maximum limit; user logging uses this specified number as a minimum.

Specifying Storage Format

Devices on the HP 3000 can transmit information in ASCII (American Standard Code for Information Interchange) and/or binary code, depending on the device.

For example, a line printer handles ASCII formatted data, while a disk can transmit and store data in either format. You can use appropriate optional parameters in `HPFOPEN/FOPEN` to specify the code (ASCII or binary) in which a new file is to be recorded when it is written to a device that supports both codes.

NOTE It is even possible to transmit and store data in EBCDIC, as long as the application program or subsystem (`FCOPY`, for example) handles the decoding/encoding. EBCDIC is not handled automatically by MPE/iX.

With many devices, there is no restriction on the data actually transferred to or from the file; you can write ASCII data to a binary file, or binary data to an ASCII file. You can specify the type of code that you want, or accept the MPE/iX default for the device that you are using.

When the allocated record space is not filled by data, MPE/iX pads the unused space with a fill character instead of good data. If you accessed this unused portion of a record (for example, with the inhibit buffering option set to `NOBUF`), you would find in the unallocated record space the fill character specified at file creation.

The fill character may be different depending upon the mechanism you used to create the file. If you create the file with `FOPEN` or `BUILD`, MPE/iX pads an ASCII file with blanks and a binary file with zeros. If you create the file with `HPFOPEN`, MPE/iX pads the file with the fill character specified by the *`fill character option`* (if not specified, the default fill character for files created with `HPFOPEN` is blanks for ASCII files and NULL characters for binary files). Examples of ASCII files on the HP 3000 include program source files, general text and document files, and MPE/iX stream files containing MPE/iX commands. Examples of binary files include program files containing linked object code, and application data files.

3 Specifying a File Designation

The name by which a program recognizes your file is its **formal file designator**. This is the file name that is coded into the program, along with the program's specifications for the file.

The formal file designator is the name by which your program recognizes the file, but there must also be a means by which the file system can recognize it, allowing it to be referenced by various commands and programs. This chapter discusses the various ways MPE/iX allows you to designate a name for your file.

As you read this chapter, keep these considerations in mind:

- How will the file be referenced?
- How will the file be used?
- How does MPE/iX associate formal and actual file designators?

MPE/iX File Designators

The file system recognizes two general classes of files:

- user-defined files, which you or other users define, create, and make available for your own purposes
- system-defined files, which the file system defines and makes available to all users to indicate standard input/output devices

These files are distinguished by the file names and other descriptors (such as group or account names) that reference them, as discussed below. You may use both the file name and descriptors, in combination, as either formal designators within your programs or as actual designators that identify the file to the system. Generally, however, most programmers use only arbitrary names as formal designators, and then equate them to appropriate actual file designators at run time. In such cases, the formal designators (user file names) contain from one to eight alphanumeric characters, beginning with a letter; the actual designators include a user or system file name, optionally followed by a group name, account name, and/or security lockword, all separated by appropriate delimiters. This technique facilitates maximum flexibility with respect to file references.

User-Defined Files

You can reference any user-defined file by writing its name and descriptors in the *filereference* format, as follows:

filename[/*lockword*][.*groupname*][.*accountname*]

In no case must any file designator written in the *filereference* format exceed 35 characters, including delimiters.

When you reference a file that belongs to your logon account and group, you need only use the *filereference* format in its simplest form, which includes only a file name that may range from one to eight alphanumeric characters, beginning with a letter (unless, of course, the file has a lockword, in which case you must specify the lockword and a delimiter). In the following examples, both formal and actual designators appear in this format:

```
         Formal designator
          |
FILE ALPHA=BETA     <---- Actual designator
FILE REPORT=OUTPUT
FILE X=AL126797
FILE PAYROLL=SELFL
```

A file reference is always qualified, in the appropriate directory, by the names of the group and account to which the file belongs, so you need ensure only that the file's name is unique within its group. For instance, if you create a file named FILX under GROUPA and ACCOUNT1, the system recognizes your file as FILX.GROUPA.ACCOUNT1; a file with the same file name, created under a different group, could be recognized as FILX.GROUPB.ACCOUNT1.

File groups serve as the basis for your local file references; thus, when you log on, if the default file system file security provisions are in effect, you have unlimited access to all files assigned to your logon group and your home group. Furthermore, you are permitted to read, and execute programs residing in, the public group of your logon account. This group, always named PUB, is created under every account to serve as a common file base for all users of the account. In addition, you may read and execute programs residing in the PUB group of the System Account. This is a special account available to all users on every system, always named SYS.

When you reference a file that belongs to your logon account, but not to your logon group, you must specify the name of the file's group within your reference. In this form of the *filereference* format, the group name appears after the file name, separated from it by a period. Embedded blanks within the file or group names, or surrounding the period, are prohibited. As an example, suppose your program references a file under the name LEDGER, which is recorded in the system by the actual designator GENACCT. This file belongs to your home group, but you are logged on under another group when you run the program. To access the file, you must specify the group name as follows:

```
FILE LEDGER=GENACCT.XGROUP   <------- Group name
RUN MYPROG                   <------- Program file (in logon group)
```

As another example, suppose that you are logged on under the group name XGROUP but wish to reference a file named X3 that is assigned to the Public Group of your account. If your program refers to this file by the name FILLER, you would enter:

```
FILE FILLER=X3.PUB
```

When you reference a file that does not belong to your logon account, you must use an even more extensive form of the *filereference* format. With this form, you include both group name and account name. The account name follows the group name, and is separated from it by a period. Embedded blanks are not permitted. As an example, suppose you are logged on under the account named MYACCT but wish to reference the file named GENINFO in the public group of the system account. Your program references this file under the formal designator GENFILE. You would enter:

```
FILE GENFILE=GENINFO.PUB.SYS
```

A file reference that includes the file name, group, and account is called a **fully qualified file name**.

NOTE You can create a new file only within your logon account; therefore, if you wish to have a new file under a different account, you log on to the other account and create the file in that account and group.

In summary, remember that if you do not supply a group name or account name in your *filereference*, MPE/iX supplies the defaults of the group and account in which you are currently logged on.

Lockwords

When you create a disk file, you can assign to it a lockword that must thereafter be supplied (as part of the *filereference* format) to access the file in any way. This lockword is independent of, and serves in addition to, the other file system security provisions governing the file.

You assign a lockword to a new file by specifying it in the *filereference* parameter of the BUILD command or the *formaldesignator* parameter of the HPFOPEN/FOPEN intrinsic used to create the file. For example, to assign the lockword SESAME to a new file named FILEA, you could enter the following BUILD command:

```
BUILD FILEA/SESAME  <---- Lockword
```

From this point on, whenever you, or another user, reference the file in an MPE/iX command or HPFOPEN/FOPEN intrinsic, you must supply the lockword. It is important to remember that you need the lockword even if you are the creator of the file. Lockwords, however, are required only for old files on disk.

When referencing a file protected by a lockword, supply the lockword in the following manner:

- In batch mode, supply the lockword as part of the file designator (*filereference* format) specified in the FILE command or HPFOPEN/FOPEN intrinsic call used to establish access to the file. Enter the lockword after the file name, separated from it by a slash mark. Neither the file name nor the lockword should contain embedded blanks.

In addition, the slash mark (/) that separates these names should not be preceded or followed by blanks. The lockword may contain from one to eight alphanumeric characters, beginning with a letter. If a file is protected by a lockword and you fail to supply that lockword in your reference, you are denied access to the file. In the following example, the old disk file XREF, protected by the lockword OKAY, is referenced:

```
FILE INPUT=XREF/OKAY     <---- Lockword
```

- In session mode, you can supply the lockword as part of the file designator specified in the FILE command or HPFOPEN/FOPEN intrinsic call that establishes access to the file, using the same syntax rules described above. If a file is protected by a lockword and you fail to supply it when you open the file, the file system interactively requests you to supply the lockword as shown in the example below:

```
LOCKWORD: YOURFILE.YOURGRP.YOURACCT?
```

Always bear in mind that the file lockword relates only to the ability to access files, and not to the account and group passwords used to log on. Three examples of FILE commands referencing lockwords are shown below; the last command illustrates the complete, fully qualified form of the *filereference* format.

```
FILE AFILE=GOFILE/Z22           <---- Lockword
FILE BFILE=FILEM/LOCKB.GRO7
                         |
                         |----  Lockwords
                         |
FILE CFILE=PAYROLL/X229AD.GROPN.ACCT10
```

A file may have only one lockword at a time. You can change or remove the lockword by using the RENAME command or the FRENAME intrinsic. You can also initially assign a lockword to an existing file with this command or intrinsic.

To accomplish these tasks, you must be the creator of the file.

Backreferencing files

Once you establish a set of specifications in a FILE command, you can apply those specifications to other file references in your job or session simply by using the file's formal designator, preceded by an asterisk (*), in those references. For example, suppose you use a FILE command to establish the specifications shown below for the file FILEA, used by program PROGA. You then run PROGA. Now you wish to apply those same specifications to the file FILEB, used by PROGB, and run that program. Rather than specify all those parameters again in a second FILE command, you can simply use FILE to equate the FILEA specifications to cover FILEB, as follows:

FILE FILEA;DEV=TAPE;REC=-80,4,V;BUF=4	Establishes specifications.
RUN PROGA	Runs program A.
FILE FILEB=*FILEA	Backreferences specifications for FILEA.
RUN PROGB	Runs program B.

This technique is called **backreferencing** files, and the files to which it applies are

sometimes known as **user predefined files**. Whenever you reference a predefined file in a file system command, you must enter the asterisk before the formal designator if you want the predefinition to apply.

System-Defined Files

System-defined file designators indicate those files that the file system uniquely identifies as standard input/output devices for jobs and sessions. These designators are described in Table 3-1. on page 77 When you reference them, you use only the file name; group or account names and lockwords do not apply.

Table 3-1. System-Defined File Designators

FILE\DESIGNATOR (NAME)	DEVICE/FILE REFERENCED
$STDIN	The standard job or session input device from which your job/session is initiated. For a session, this is always a terminal. For a job, it may be a disk file or other input device. Input data images in this file should not contain a colon in column 1, because this indicates the end-of-data. (When data is to be delimited: use the EOD command, which performs no other function.)
$STDINX	Same as $STDIN except that MPE/iX command images (those with a colon in column 1) encountered in a data file are read without indicating the end-of-data; however, the commands :EOD and EOF (and in batch jobs, the commands JOB, EOJ and DATA) are exceptions that always indicate end-of-data, but are otherwise ignored in this context--they are never read as data. $STDINX is often used by interactive subsystems and programs to reference the terminal as an input file.
$STDLIST	The standard job or session listing device, nearly always a terminal for a session and a printer for a batch job.
$NULL	The name of a nonexistent ghost file that is always treated as an empty file. When referenced as an input by a program, that program receives an end-of-data indication upon each access. When referenced as an output file, the associated write request is accepted by MPE/iX but no physical output is actually done. Thus, $NULL can be used to discard unneeded output from a running program.

As an example of how to use some of these designators, suppose you are running a program that accepts input from a file programmatically defined as INFILE and directs output to a file programmatically defined as OUTFILE. Your program specifies that these are disk files, but you wish to respecify these files so that INFILE is read from the standard input device and OUTFILE is sent to the standard listing device.

You could enter the following commands:

```
FILE INFILE=$STDIN
FILE OUTFILE=$STDLIST
RUN MYPROG
```

Input/Output sets

All file designators can be classified as those used for input files (Input Set), or those used for output files (Output Set). For your convenience, these sets are summarized in Table 3-2. on page 78 and Table 3-3. on page 78

Table 3-2. Input Set

File Designator	Function/Meaning
$STDIN	Job/session input device.
$STDINX	Job/session input device with commands allowed.
$OLDPASS	Last $NEWPASS file closed. Discussed in the following pages.
$NULL	Constantly empty file that returns end-of-file indication when read.
*formaldesignator	Back reference to a previously defined file.
filereference	File name, and perhaps account and group names and lockword. May be a temporary file created in current job/session, created and saved in any job/session.

Table 3-3. Output Set

File Designator	Function/Meaning
$STDLIST	Job/session list device.
$OLDPASS	Last file passed. Discussed in following pages.
$NEWPASS	New temporary file to be passed. Discussed in the following pages.
$NULL	Constantly empty files that returns end-of-file indication when read.
*formaldesignator	Back reference to a previously defined file.
filereference	File name, and perhaps account and group names and lockword. Unless you specify otherwise, this is a temporary file residing on disk that is destroyed on termination of the creating program. If closed as a temporary file, it is purged at the end of the job/session. If closed as a permanent file, it is saved until you purge it.

An input file and a list file are said to be interactive if a real-time dialog can be established between a program and a person using the list file as a channel for programmatic requests, with appropriate responses from a person using the input file. For example, an input file and a list file opened to the same teleprinting terminal (for a session) would constitute an interactive pair. An input file and a list file are said to be duplicative when input from the

former is duplicated automatically on the latter. For example, input from a magnetic tape device is printed on a line printer. You can determine whether a pair of files is interactive or duplicative with the FRELATE intrinsic call. (The interactive/duplicative attributes of a file pair do not change between the that time the files are opened and the time they are closed.)

The FRELATE intrinsic applies to files on all devices. To determine if the input file INFILE and the list file LISTFILE are interactive or duplicative, you could issue the following FRELATE intrinsic call:

```
ABLE := FRELATE(INFILE,LISTFILE);
```

INFILE and LISTFILE are identifiers specifying the file numbers of the two files. The file numbers were assigned to INFILE and LISTFILE when the HPFOPEN/FOPEN intrinsic opened the files.

A half-word is returned to ABLE showing whether the files are interactive or duplicative. The half-word returned contains two significant bits, 0 and 15:

if bit 15 = 1, INFILE and LISTFILE form an interactive pair

if bit 15 = 0, INFILE and LISTFILE do not form an interactive pair

if bit 0 = 1, INFILE and LISTFILE form a duplicative pair

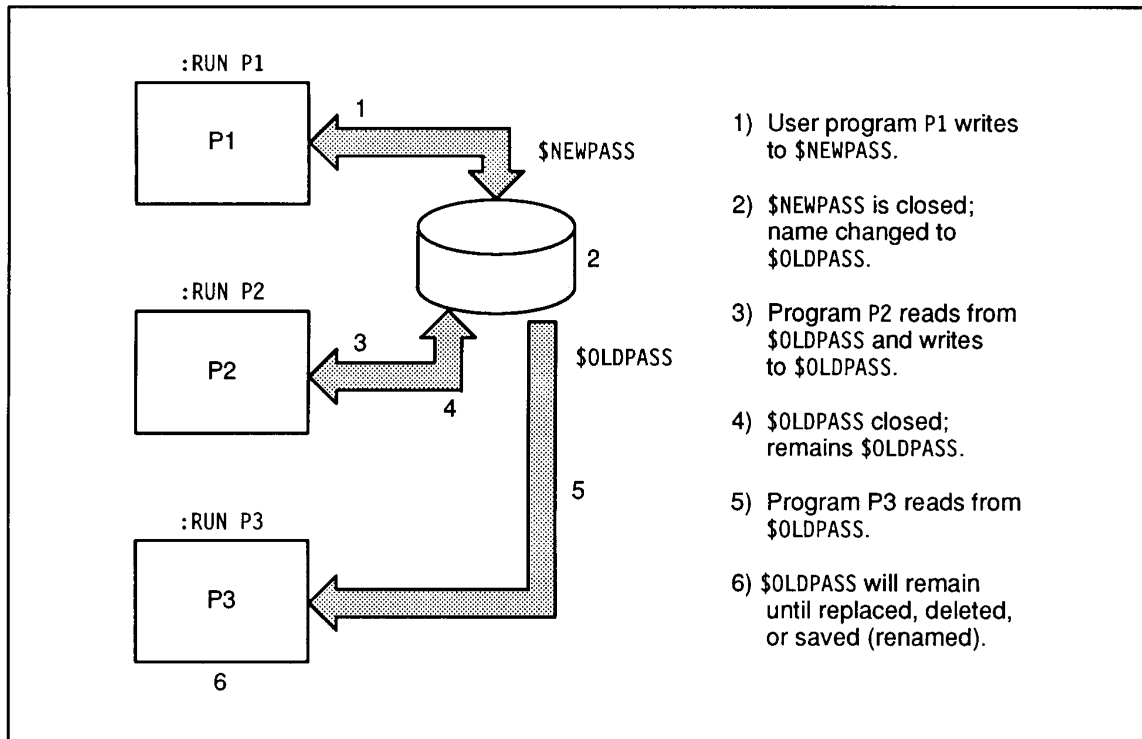
if bit 0 = 0, INFILE and LISTFILE do not form a duplicative pair

Passed files

Programmers, particularly those writing compilers or other subsystems, sometimes create a temporary disk file that can be automatically passed to succeeding MPE/iX commands within a job or session. This file is always created under the special name $NEWPASS. When your program closes the file, MPE/iX automatically changes its name to $OLDPASS and deletes any other file named $OLDPASS in the job/session temporary file domain. From this point on, your commands and programs reference the file as $OLDPASS. Only one file named $NEWPASS and/or one file named $OLDPASS can exist in the job/session domain at any one time.

The automatic passing of files between program runs is depicted in Figure 3-1. To illustrate how file passing works, consider an example where two programs, PROG1 and PROG2, are executed. PROG1 receives input from the actual disk file DSFILE (through the programmatic name SOURCE1) and writes output to an actual file $NEWPASS, to be passed to PROG2. ($NEWPASS is referenced programmatically in PROG1 by the name INTERFIL.) When PROG2 is run, it receives $NEWPASS (now known by the actual designator $OLDPASS), referencing that file programmatically as SOURCE2. Note that only one file can be designated for passing.

Figure 3-1. Passing Files between Program Runs

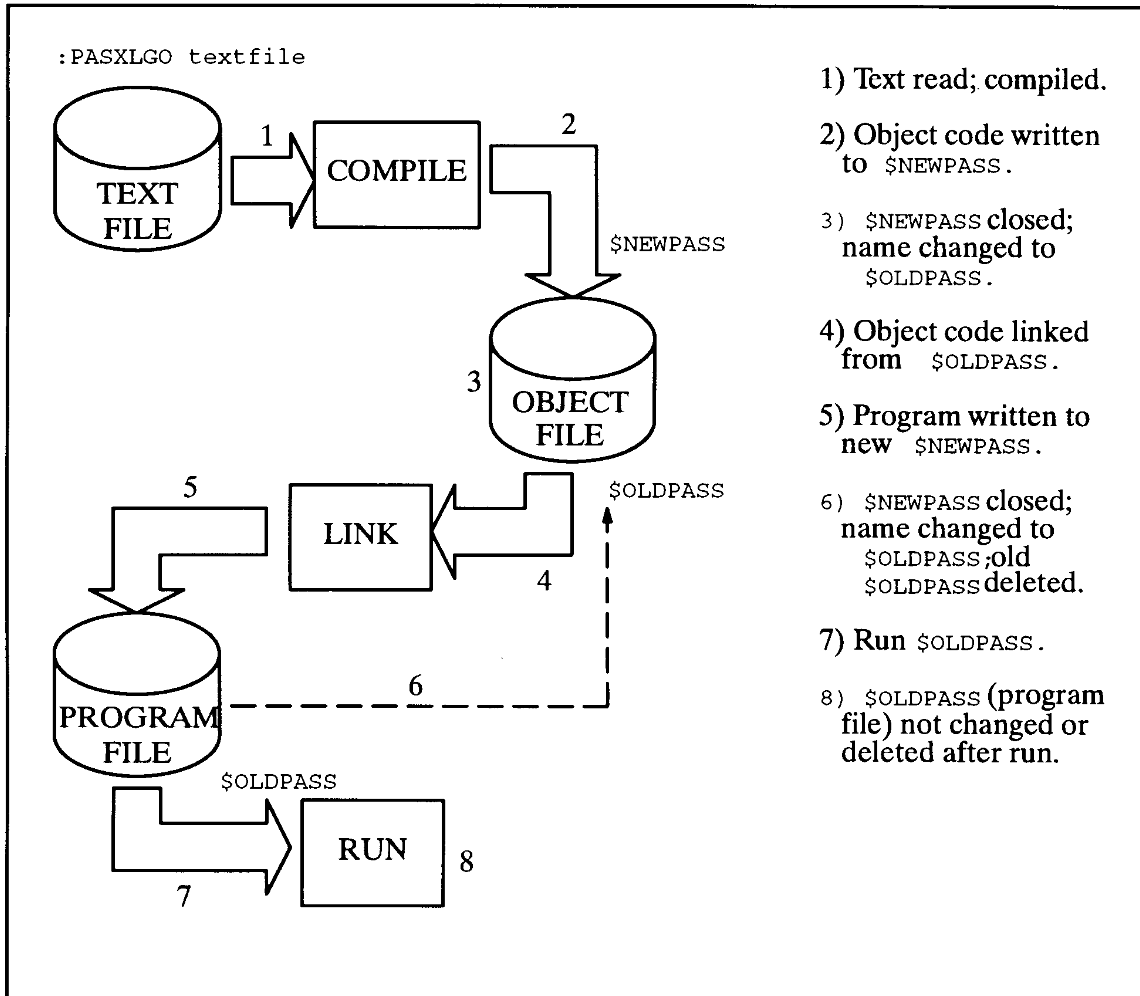



```
.
.
.
FILE SOURCE1=DSFIL
FILE INTERFIL=$NEWPASS      <---
RUN PROG1                       |-  Same File
FILE SOURCE2=$OLDPASS       <---
RUN PROG2
.
.
.
```

A program file must pass through several steps as it is executed; passed files are most frequently used between these steps. A program file must be compiled and linked before it is executed. By default, the compiled form of a text file is written to $NEWPASS. When the compiler closes $NEWPASS, its name is changed to $OLDPASS; it is this file that is linked for execution. The linked form of the program file is written to a new $NEWPASS, which is renamed $OLDPASS when the file is closed; the old $OLDPASS is deleted. Now, this file is ready to be executed. This $OLDPASS may be executed any number of times, until it is overwritten by another $OLDPASS file.

The steps that a program takes as it is run are depicted in Figure 3-2..

Figure 3-2. Passing Files within a Program Run



`$NEWPASS` and `$OLDPASS` are specialized disk files with many similarities to other disk files. Comparisons of `$NEWPASS` to new files, and `$OLDPASS` to old files, are given in Table 3-4. on page 81 and Table 3-5. on page 82.

Table 3-4. New Files Versus $NEWPASS

NEW	$NEWPASS
Disk space allocated.	Disk space allocated.
Disk address put into control block.	Disk address put into control block.
Default close disposition: Deallocate space. Delete control block entry.	Default close disposition: Rename to `$OLDPASS`. Save disk address in current job or session table. (Job Information Table) Delete control block entry.

Table 3-4. New Files Versus $NEWPASS

NEW	$NEWPASS
Disk address not saved (Not in any directory).	Disk address saved for future use in the current job session.

Table 3-5. Old Files Versus $OLDPASS

OLD	$OLDPASS
Directory (job temporary or system) searched for disk address	Disk address obtained from Job Information Table (JIT)
Disk address put into control block.	Disk address put into control block.
Default close disposition: Delete control block.	Default close disposition: Delete control block.
Disk address still in directory for future use.	Disk address still in JIT for future use in current job session.

Using Command Interpreter Variables and Expressions Within File Designators

Your file reference may also contain command interpreter variables and expressions that are evaluated before the file reference is parsed and validated. In the following file equation, the exclamation point (!) instructs MPE/iX to substitute the variable name `MYFILE` with the actual file designator assigned to that variable by the `SETVAR` command.

```
SETVAR MYFILE,'FILE2.MYGROUP.MYACCT'
FILE FILE1=!MYFILE
```

The HPFOPEN and FOPEN intrinsics also allow you to embed command interpreter variables and expressions in the file reference. The following file references are valid when passed as formal designators:

```
!MYFILE
!MYFILE.!HPGROUP.!HPACCT
!FILE1/![FINFO(-!FILE1",33)]
```

For more information about using command interpreter variables and expressions, refer to the *Command Interpreter Access and Variables Programmers' Guide*.

Parsing and Validating File Designators

The FPARSE intrinsic parses and validates a file designator string to determine if it is syntactically correct. You can employ this intrinsic to check a formal file designator representing a file before attempting to open the file with HPFOPEN/FOPEN. MPE/iX file designators used for the file system and two user interface commands include a remote environment ID (*envid*). This allows the user to indicate that a file is to be accessed from a remote environment established by the user with the DSLINE or REMOTE HELLO command. FPARSE facilitates the changes required for the file designator extension. It provides the only location within MPE/iX where file designators are parsed and syntax is checked.

The following are examples of the *items* and the *vectors* array pair. The order of entries in the *vectors* array corresponds to the order of items in the *items* array. Each 32-bit entry in the *vectors* array returns the byte offset of the item in the first half-word, and the length in bytes of the item in the second half-word. However, the last entry of the *vectors* array has a different meaning from that of the other entries: the second half-word gives the total length of the file string, and the first half-word gives a system file code when applicable.

In Figure 3-3. the file string is
"FILENAME/LOCKWORD.GROUP.ACCOUNT:ANIMAL.INDDCL.HPBCG":

Figure 3-3. Illustration of FPARSE Usage

items ARRAY (bits 0–15)	array element	*vectors* ARRAY (bits 0–15)	*vectors* ARRAY (bits 16–31)
1	(0)	0	8
5	(1)	32	19
3	(2)	18	5
4	(3)	24	7
2	(4)	9	8
0	(5)	0	51

The *items* array, as illustrated above, can be listed in any order or can be left unspecified if not required.

In Figure 3-4., below, the file string is "*FILENAME:ANIMAL":

Figure 3-4. Illustration of FPARSE Usage

items ARRAY (bit positions 0123456789012345)

array element	items
(0)	1
(1)	2
(2)	3
(3)	4
(4)	5
(5)	0

vectors ARRAY (bit positions 01234567890123456789012345678901)

array element	bits 0–15	bits 16–31
(0)	1	8
(1)	0	0
(2)	0	0
(3)	0	0
(4)	10	6
(5)	0	16

In Figure 3-5., below, the file string is `"$OLDPASS"`:

Figure 3-5. Illustration of FPARSE Usage

items ARRAY (bit positions 0123456789012345)

array element	items
(0)	1
(1)	2
(2)	3
(3)	4
(4)	5
(5)	0

vectors ARRAY (bit positions 01234567890123456789012345678901)

array element	bits 0–15	bits 16–31
(0)	0	8
(1)	0	0
(2)	0	0
(3)	0	0
(4)	0	0
(5)	0	8

Note that "$" is a special exception to the rules of file names and is considered part of the file name, unlike "*", which is not.

4 Specifying a File Domain

One way to classify a file is on the basis of its domain. A file can be permanent or temporary, or it may exist only to one particular process. The file system maintains separate directories to record the location of temporary files and permanent files. Of course, there is no file system directory for files that exist only to their creating process (new files).

In this chapter, we will address the following questions:

- What do the various domains mean?
- Can a file's domain be changed?
- How can the files in various domains be listed?

New Files

When you create a file, you can indicate to the file system that it is a new file; it has not previously existed. Space for this file has not yet been allocated. As a new file, it is known only to the program that creates it, and exists only while the program is being executed. When the program concludes, the file simply vanishes, unless you take actions to retain it.

Temporary Files

A temporary file is one that already exists, but that is known only to the job or session that created it. Some or all of the space for a temporary file has already been allocated, and its physical characteristics have already been defined. A file in this domain is considered a job temporary file; it was created for some specific purpose by its job or session and may not be needed when the job or session concludes. Like a new file, it vanishes when its creating job or session is over.

Permanent Files

A permanent file exists as a file in the system file domain. Its existence is not limited to the duration of its creating job or session, and depending on security restrictions, it may be accessed by jobs or sessions other than the one that created it. Some or all of the space for a permanent file has already been allocated, and its physical characteristics have been defined.

Table 4-1. Features of New, Temporary, and Permanent Files

New Files	Temporary Files	Permanent Files
Exists only to creating process.	Exists as job temporary file.	Exists as permanent file in system.
Space not allocated yet.	Space (some or all) already allocated.	Space (some or all) already allocated.
Physical characteristics not previously defined.	Physical characteristics defined.	Physical characteristics defined.
Known only to creating job or session.	Known only to creating job or session.	Known system-wide.
Exists only for duration of program execution.	Exists only for duration of creating job/session.	Permanent.

Table 4-2. File Domains Permitted

Device Type	Domain Permitted
Disk	new, temporary, or permanent
Magnetic Tape Device	new * or permanent
Synchronous Single-Line Controller	new * or permanent
Programmable Controller	new * or permanent
Terminal	new * or permanent
Line Printer	new *
Plotter	new *

The features of new, temporary and permanent files are listed in Table 4-1. on page 88

In some cases, the domain you can specify for a file may be restricted by the type of device on which the file resides. The domains permitted are summarized in Table 4-1. on page 88

NOTE * When you specify a file domain using HPFOPEN, you should open only disk files with the *domain option* set to NEW. Device files can be opened with the

domain option set to NEW (to maintain compatibility with with MPE V/E), but a warning is returned in the *status* parameter.

Changing Domains

A file need not always stay in the same domain. Any disk file can be made permanent, or can be deleted when it has served its purpose. The disposition parameter of the FCLOSE intrinsic can specify a different domain for a file as it closes, or the FILE command can be used to change the domain of a file. The DEL, TEMP, and SAVE parameters determine what happens to the file when it is closed. For details about how the FCLOSE intrinsic handles file domain disposition, refer to chapter 6, "Closing a File".

A file in any domain may be deleted if the DEL parameter is used in a file equation. For example, suppose that you have a permanent file named OLDFL, and want to delete it after its next use. Before running the program that uses OLDFL, enter:

```
FILE OLDFL;DEL
```

The file may now be opened in your program, and when the program closes the file, it is deleted. If OLDFL were a new or temporary file, it would be deleted in the same way.

New files may be made temporary if the TEMP parameter is used in a file equation. If you are about to create a file named NEWFL, and wish it to remain as a temporary file after it is used, enter:

```
FILE NEWFL,NEW;TEMP
```

After the file is created in your program and is closed, the file system maintains it as a temporary file. If you wish to keep a new or temporary file as a permanent file after it is used, use the SAVE parameter in a file equation. If you have a temporary file named TEMPFL, and you want it to be kept as an permanent file in the system, enter:

```
FILE TEMPFL,OLDTEMP;SAVE
```

TEMPFL is kept as a permanent file, so it will not be lost when your job or session concludes.

File equations are useful for determining the disposition of files when the files have been programmatically accessed and closed. By using the MPE/iX SAVE command, you can keep a temporary file as permanent without opening and closing the file. If you want to keep a temporary file named TEMPDATA, but do not need to use it in a program at this time, enter:

```
SAVE TEMPDATA
```

and the file system immediately identifies it as a permanent file. If there were a lockword associated with TEMPDATA, you would be prompted for it. You can use the SAVE command to keep $OLDPASS and assign it a name for future reference by entering:

```
SAVE $OLDPASS,filename
```

where *filename* is any name that you choose.

For more information about the FILE and SAVE commands, consult the *MPE/iX Commands Reference Manual*.

Searching File Directories

There are two directories with addresses of files: the temporary file directory (job file domain) for the addresses of temporary files and the permanent file directory (system file domain) for the addresses of permanent files. There is no directory for new files. When both directories are searched for a file address (for example, when you open a file with the *domain option* set to OLD), the temporary file directory is searched first.

Listing Files

To obtain a list of your permanent files, enter the LISTFILE command. Use the LISTFILE...;TEMP command to list your temporary files and the LISTEQ command to list FILE equations. The LISTFILE, LISTFILE...;TEMP, and LISTEQ commands are discussed in detail in the *Getting System Information Programmer's Guide* and in the *MPE/iX Command Reference Manual*.

5 Opening a File

Before your program can read, write, or otherwise manipulate a file, the program must initiate access to that file by opening it with the HPFOPEN/FOPEN intrinsic call. This call applies to both disk files and device files. This chapter discusses how you can use HPFOPEN to open various types of files supported by MPE/iX. Examples of program segments are provided to illustrate HPFOPEN calls.

This chapter is divided into the following subjects:

- how the file system opens a file
- which to use: HPFOPEN or FOPEN?
- opening a disk file
- opening a system-defined file
- opening a device file

How the File System Opens a File

When you open a file, HPFOPEN/FOPEN establishes a communication link between the file and your program by

- Determining the device on which the file resides.
- Allocating to your process the device on which the file resides. Disk files generally can be shared concurrently among jobs and sessions. Magnetic tape and unit-record devices are generally allocated exclusively to the requesting job or session.

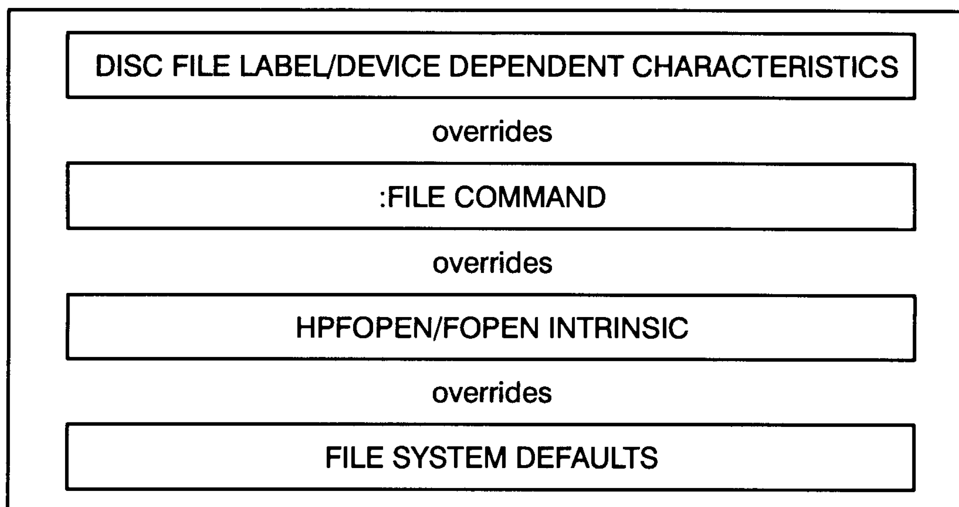
  If the file resides on a nonshareable device (such as magnetic tape) and you have nonshareable device (ND) capability, HPFOPEN/FOPEN determines whether the system operator must approve allocation of the device (such as an unlabeled magnetic tape) or provide a particular media (such as a specific volume for a labeled magnetic tape request or special forms for a line printer). If so, HPFOPEN/FOPEN requests the system operator to respond appropriately.

  Different processes within the same job may open and have concurrent access to a file on the same magnetic tape or unit-record device if the file has been opened with *multiaccess option* set; however, this device cannot be accessed by another job until all accessing processes in this job have issued a corresponding FCLOSE call.
- Verifying your right to access the file under the security provisions existing at the account, group, and file levels.

- Determining that the file has not been allocated exclusively to another process (by the exclusive option in an HPFOPEN/FOPEN call issued by that process).
- Processing user labels (for files on disk). For new files on disk, HPFOPEN/FOPEN specifies the number of user labels to be written.
- Constructing the control blocks required by MPE/iX for this particular access of the file. The information in these blocks is derived by merging specifications from four sources, listed below in descending order of precedence (and illustrated in Figure 5-1.)
  1. The file label, obtainable only if the file is an old file on disk; otherwise, device-dependent characteristics applicable to the nonshareable device. This information overrides information from any other source.
  2. The parameter list of a previous FILE command referencing the same formal file designator named in this HPFOPEN/FOPEN call, if such a command was issued in this job or session. This is only true, if file equations were not disallowed.
  3. The parameter list of this HPFOPEN/FOPEN intrinsic call.
  4. System default values provided by MPE/iX (when values are not obtainable from the above sources).

When information from one of these four sources conflicts with that from another, preempting takes place according to the order of precedence illustrated in Figure 5-1. To determine the specifications actually taking effect, you can call the FFILEINFO intrinsic. Certain sources do not always apply or convey all types of information. For example, no file label exists when a new file is opened, and so all information must come from the last four sources above.

Figure 5-1. File System Hierarchy of Overrides



Since the physical characteristics of a disk file cannot be changed after it has been created, it makes sense that the file label would take precedence over information from any other source. Likewise, when a device file is opened, device-dependent characteristics override information from any other source.

When the HPFOPEN/FOPEN intrinsic is executed, it returns to your program a file number. If the file is opened successfully, the file number returned is a positive integer. At this point, the file is ready to be accessed with system intrinsics (for example, FREAD or FWRITE). If the file cannot be opened, the file number returned is zero, and the intrinsic returns an error condition.

If your process issues more than one HPFOPEN/FOPEN call for the same file before it is closed, this results in multiple, logically separate accesses of that file, and MPE/iX returns a unique file number for each such access. Also, MPE/iX maintains a separate logical record pointer (indicating the next sequential record to be accessed) for each access where you did not request or permit the *multiaccess option* at HPFOPEN/FOPEN time.

Which to Use: HPFOPEN or FOPEN

The HPFOPEN intrinsic is the recommended intrinsic for creating and opening files on an MPE/iX-based computer system. HPFOPEN is designed to be more flexible and extendible than the FOPEN intrinsic. In addition, HPFOPEN's available options are a superset of the options available through FOPEN. For example, mapped access is available through HPFOPEN but not through FOPEN.

NOTE One HPFOPEN option, the *file privilege option*, when used to set a new file's privilege level to other than 3 (least-privileged, or user level), disallows all subsequent access of that file by the FOPEN intrinsic. (For compatibility reasons, FOPEN can only access a file whose file privilege level is three.)

You should use the FOPEN intrinsic only if you are planning to migrate your application between MPE/iX-based systems and MPE V/E-based systems. HPFOPEN is not supported on an MPE V/E-based system.

The examples in this chapter illustrate the use of the HPFOPEN intrinsic. The HPFOPEN intrinsic uses an *itemnum, item* pair convention for optional parameter passing. The *itemnum* and *item* parameters are paired, where the *n*th *itemnum* is immediately followed in the parameter list by the *n*th *item*. The *itemnum* parameter passes by value an integer that the intrinsic uses to define the meaning and expected data type of the value passed by reference in the corresponding *item* parameter.

For details on HPFOPEN and FOPEN parameters, refer to the respective intrinsics descriptions in the *MPE/iX Intrinsics Reference Manual*.

Opening a Disk File

Disk files are files residing on volumes (disk packs). Disk files are immediately accessible by the system and potentially shareable by several jobs or sessions at the same time. The

following examples show how you can use the HPFOPEN intrinsic to open a disk file:

- "Opening a new disk file" shows an example of an HPFOPEN call that creates a new disk file (see example 5-1).
- "Opening a permanent disk file" shows an example of an HPFOPEN call that opens a permanent disk file that is to be shared among multiple concurrent accessors (see example 5-2).

Opening a new disk file

Example 5-1 is an HP Pascal/iX code segment containing an HPFOPEN intrinsic call that opens a new disk file to be used with a text editor. The text editor accesses only standard ASCII text files with fixed-length records, each record 80 bytes in size.

Knowing this, you can specify the appropriate HPFOPEN options, and allow others (specifically, *domain option*, *record format option*, and *file type option*) to default to the desired characteristics. Note that the HPFOPEN *final disposition option* is specified to indicate that the file is to be saved as a temporary file at close time.

Example 5-1. Opening a New Disk File

```
procedure open_new_disk_file;

const
   formal_designator_option = 2;     {defines HPFOPEN itemnum 2  }
   record_size_option       =19;     {defines HPFOPEN itemnum 19 }
   final_disposition_option =50;     {defines HPFOPEN itemnum 50 }
   ASCII_binary_option      =53;     {defines HPFOPEN itemnum 53 }

type
   pac80  = packed array [1..80] of char;

var
   file_num    : integer; {required HPFOPEN filenum parameter     }
   status      : integer; {returns info if error/warning occurs}

   file_name   : pac80;              {declares HPFOPEN itemnum 2  }
   line_len    : integer;            {declares HPFOPEN itemnum 19 }
   save_perm   : integer;            {declares HPFOPEN itemnum 50 }
   ascii       : integer;            {declares HPFOPEN itemnum 53 }

begin
   file_num    :=0;
   status      :=0;
   file_name   :='&myfile/lock.mygroup&'; (filereference format}
   line_len    :=80;          {maximum record/line length     }
   save_temp   :=2;           {make temp file at close        }
   ascii       :=1;           {label indicates ASCII code     }
    HPFOPEN (file_num, status,
      formal_designator_option,file_name, {formal designator option}
      record_size_option, line_len,       {record size option      }
      final_disp_option, save_temp,      {final disposition option }
```

```
        ASCII_binary_option, ascii            {ASCII/binary option     }
        );

      if status <> 0 then handle_file_error (file_num, status);
   end;
```

If the HPFOPEN call is successful, a positive integer value is returned in file_num, and status returns a value of zero. The new disk file is now open and can be accessed with system intrinsics. If an error or warning condition is encountered by HPFOPEN, status returns a nonzero value, thus invoking the error-handling procedure handle_file_error.

In appendix A, "HP Pascal/iX Program Examples," Example A-1 uses a similar procedure to open a new disk file. For more information about HPFOPEN parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Opening a permanent disk file

Example 5-2 is an HP Pascal/iX code segment containing an HPFOPEN intrinsic call that opens a permanent disk file that is to be shared among multiple concurrent accessors. Note the use of the *dynamic locking option* to enable the use of file-locking intrinsics (FLOCK and FUNLOCK) with this file. The file is opened update access to allow opening the file with Read/Write access without affecting the current EOF. Thus, current data in the file is retained.

Example 5-2. Opening a Permanent Disk File

```
procedure open_permanent_disk_file;

const
   formal_designator_option = 2;     {defines HPFOPEN itemnum 2  }
   domain_option            = 3;     {defines HPFOPEN itemnum 3  }
   access_type_option       =11;     {defines HPFOPEN itemnum 11 }
   dynamic_locking_option   =12;     {defines HPFOPEN itemnum 12 }
   exclusive_option         =13;     {defines HPFOPEN itemnum 13 }
   ASCII_binary_option      =53;     {defines HPFOPEN itemnum 53 }

type
   pac80  = packed array [1..80] of char;

var
   file_num    : integer; {required HPFOPEN filenum parameter   }
   status      : integer; {returns info if error/warning occurs}

   file_name   : pac80;              {declares HPFOPEN itemnum 2  }
   permanent   : integer;            {declares HPFOPEN itemnum 3  }
   update      : integer;            {declares HPFOPEN itemnum 11 }
   lockable    : integer;            {declares HPFOPEN itemnum 12 }
   shared      : integer;            {declares HPFOPEN itemnum 13 }
   ascii       : integer;            {declares HPFOPEN itemnum 53 }

begin
   file_num    :=0;
   status      :=0;
   file_name   :='&datafile/![FINFO("datafile",33)].!hpgroup&';
   permanent   :=1;            {search in permanent file directory}
```

```
update      :=5;         {enable update access to file   }
lockable    :=1;         {enable dynamic locking option  }
shared      :=3;         {allow concurrent access by all }
ascii       :=1;         {label will indicate ASCII code }

 HPFOPEN (file_num, status,
   formal_designator_option,file_name,  {formaldesignator option}
   domain_option, permanent,            {domain option}
   access_type_option, update,          {access type option}
   dynamic_locking_option, lockable,    {dynamic locking option}
   exclusive_option, shared,            {exclusive option}
   ASCII_binary_option, ascii           {ASCII/binary option}
   );

   if status <> 0 then handle_file_error (file_num, status);
end;
```

The file name passed in the *formaldesignator option* contains MPE/iX command interpreter variables and expressions that are evaluated by `HPFOPEN` before the file name is parsed and evaluated. `HPFOPEN` substitutes `![FINFO(datafile",33)]` with the lockword associated with file `datafile` (if the security provisions in effect enable you to obtain the file's password). The exclamation point (!) before the variable name `hpgroup` instructs `HPFOPEN` to substitute the value of the variable in place of the variable name. For more information about using command interpreter variables and expressions, refer to the *Command Interpreter Access and Variables Programmer's Guide*.

If the `HPFOPEN` call is successful, a positive integer value is returned in `file_num`, and `status` returns a value of zero. The file is now open and can be accessed with file system intrinsics. If an error or warning condition is encountered by `HPFOPEN`, `status` returns a nonzero value, thus invoking the error-handling procedure `handle_file_error`.

In appendix A "HP Pascal/iX Program Examples," Example A-5 uses a similar procedure to open a permanent disk file. For more information about `HPFOPEN` parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Opening a System-Defined File

System-defined file designators indicate those files that the file system uniquely identifies as standard input/output devices for jobs and sessions. System-defined files are `$STDIN`, `$STDINX`, `$STDLIST`, `$NEWPASS`, `$OLDPASS`, and `$NULL`. You cannot redefine characteristics for these files once the process executing your code has been created, nor can you backreference a file equation to redefine the characteristics for a system-defined file designator. For more information about system-defined files, refer to Chapter 3, "Specifying a File Designator".

The following examples show two different ways that you can open system-defined files using the indicated `HPFOPEN` options:

- "Opening `$STDIN`" shows an example of an `HPFOPEN` call that uses the *designator option* to open the job or session standard input device (see Example 5-3).
- "Opening `$STDLIST`" shows an example of an `HPFOPEN` call that uses the *formaldesignator option* to open the job or session standard list device (see example 5-4).

Opening $STDIN

Example 5-3 is an HP Pascal/iX code segment containing an `HPFOPEN` intrinsic call that uses only the *designator option* to open the system-defined file `$STDIN`. `$STDIN` is the file designator associated with your job or session's standard input device. For an interactive session, `$STDIN` is always a terminal keyboard. For a batch job, `$STDIN` may be a disk file or other input device. You can also open a system-defined file using only the `HPFOPEN` *formaldesignator option* (illustrated in Example 5-4).$STDIN, opening:files |

Example 5-3. Opening $STDLIST Using HPFOPEN *designator option*

```
procedure open_standard_input_device

const
   designator_option    = 5;     {defines HPFOPEN itemnum 5}

var
   inputfile_num  : integer;  {required HPFOPEN filenum parameter}
   status         : integer;  {returns info if error/warning occurs}

   designator     : integer;  {declares HPFOPEN item 5           }

begin
   inputfile_num  := 0;
   status         := 0;
   designator     := 4;       {Specifies $STDIN                     }

   HPFOPEN  (inputfile_num, status,
             designator_option, designator,  {HPFOPEN designator option}
             );

   if status <>0 then handle_file_error (inputfile_num, status);
end;
```

If the HPFOPEN call is successful, a positive integer value is returned in inputfile_num, and status returns a value of zero. The file is now open and can be read from. If an error or warning condition is encountered by HPFOPEN, status returns a nonzero value, thus invoking the error-handling procedure handle_file_error. For more information about HPFOPEN parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Opening $STDLIST

Example 5-4 is an HP Pascal/iX code segment containing an HPFOPEN intrinsic call that uses the *formaldesignator option* to open the system-defined file $STDLIST.

$STDLIST is the file designator associated with your job or session's standard list device. For an interactive session, $STDLIST is nearly always a terminal screen. For a batch job, $STDLIST is usually a line printer. You can also open $STDLIST using the HPFOPEN *designator option* (illustrated in Example 5-3).

Example 5-4. Opening $STDLIST Using HPFOPEN *formaldesignator option*

```
procedure open_standard_list_device

const
   formal_designator_option   = 2;     {defines HPFOPEN itemnum 2}

type
   pac80  = packed array [1..80] of char;

var
   listfile_num   : integer;  {required HPFOPEN filenum parameter}
   status         : integer;  {returns info if error/warning occurs   }

   file_name      : pac80;    {declares HPFOPEN item 2            }

begin
   listfile_num   := 0;
   status         := 0;
   file_name      := '$stdlist'; {Specifies system-defined file      }
                                 {Blank is used as delimiter         }

   HPFOPEN  (listfile_num, status,
             formal_designator_option, file_name,
                                         {formaldesignator option}
             );

   if status <>0 then handle_file_error (listfile_num, status);
end;
```

If the HPFOPEN call is successful, a positive integer value is returned in listfile_num, and status returns a value of zero. The standard list device is now open and can be written to. If an error or warning condition is encountered by HPFOPEN, status returns a nonzero value, thus invoking the error-handling procedure handle_file_error.

In appendix A, "HP Pascal/iX Program examples," example A-1 uses a similar procedure to open $STDLIST. For more information about HPFOPEN parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Opening a Device File

Device files are files that are currently being input to or output from a nonshareable device (any peripheral device except a disk). Because all file open operations are accomplished through the file system, you can open files on very different devices in a standard, consistent way, using the HPFOPEN or FOPEN intrinsics. Furthermore, the name and characteristics assigned to a file when it is defined in a program do not restrict that file to residing on the same device every time the program is run. In these cases, the file system temporarily overrides the programmatic characteristics with those characteristics required by the device.

The following topics provide you with further discussions concerning device files, as well as two program examples to illustrate how to open a magnetic tape file:

- "Device-Dependent File Characteristics" discusses those file characteristics affected by particular devices.
- "New and permanent device files" discusses the domains required by various input/output devices.
- "Opening an unlabeled magnetic tape file" shows an example of an HPFOPEN call that opens an unlabeled magnetic tape file (see example 5-5).
- "Opening a labeled magnetic tape file" shows an example of an HPFOPEN call that opens a labeled magnetic tape file (see example 5-6).

Device-dependent file characteristics

Certain physical and access characteristics for device files are restricted by the devices on which the file resides. For your convenience, device-dependent restrictions for several devices are summarized in Table 5-1. on page 99

Table 5-1. Device-Dependent Restrictions

DEVICE TYPE	RESTRICTED FILE CHARACTERISTICS
Terminal (parallel input/output device)	*record format option* = undefined-length records *block factor option* = 1 *inhibit buffering option* = NOBUF *ASCII/binary option* = ASCII
Magnetic tape drive (serial input/output device)	No restrictions
Line printer/plotter (serial output device)	*domain option* = NEW *record format option* = undefined-length records *access type option* =Write only *block factor option* = 1
Laser printer (serial output device)	Initially and always spooled *access type option* =Write only All other restrictions same as for line printer

New and permanent device files

When a process accesses a device file (a file that resides on a nonshareable device), the device's attributes may override information passed in the *domain option* of the HPFOPEN/FOPEN call. Devices used for input only are considered permanent files. Devices used for output only, such as line printers, are considered new files. Serial input/output devices, such as terminals and magnetic tape drives, follow the *domain option* specification in your HPFOPEN/FOPEN call

NOTE The HPFOPEN intrinsic assumes that all files on nonshareable devices (device files) are permanent files. To maintain compatibility with MPE V/E, device files can be opened with the *domain option* specifying a new file, but a warning is returned in the *status* parameter.

When your job or session attempts to open a permanent file on a nonshareable device, MPE/iX searches for the file in the input device directory (IDD). If the file is not found, a message is transmitted to the system console requesting the system operator to locate the file by taking one of the following steps:

- Indicate that the file resides on a device that is not in auto-recognition mode. No DATA command is required; the System Operator simply allocates the device.
- Make the file available on an auto-recognizing device, and allocate that device.
- Indicate that the file does not exist on any device; in this case, your HPFOPEN/FOPEN request is rejected.

When you use the *device name option* or *device class option* of HPFOPEN/FOPEN to open a file on a nonshareable device (other than magnetic tape), you are requesting that an unused device be allocated to your job or session. The first available device is allocated to your job or session; the System Operator is not required to intervene. The device is immediately available if it is not being used by another job or session, or if is already allocated to your job or session by a previous HPFOPEN/FOPEN call.

If the device is already allocated to your job or session, you can specify that device by passing its logical device number (LDEV) in the *device name option* of HPFOPEN/FOPEN. Be certain, though, that you don't invoke a file equation that overrides the LDEV. (You can use the FFILEINFO intrinsic to determine the LDEV assigned to an opened file.)

When you use the *device name option* or *device class option* of HPFOPEN/FOPEN to open a file on a magnetic tape drive, operator intervention is usually required. The operator must make the tape available, unless the tape is already mounted and recognized by MPE/iX, it is auto-allocating, or if the tape drive is already allocated to the job or session.

Opening an unlabeled magnetic tape file

Example 5-5 is an HP Pascal/iX code segment containing an HPFOPEN intrinsic call that opens an unlabeled magnetic tape file TAPEFILE. The intrinsic call assumes that the tape drive associated with device class TAPE supports a density of 1600 bpi.

Example 5-5. Opening an Unlabeled Magnetic Tape File

```
procedure open_unlabeled_magnetic_tape_file;

const
   formal_designator_option = 2;     {defines HPFOPEN itemnum 2  }
   domain_option             = 3;     {defines HPFOPEN itemnum 3  }
   access_type_option        =11;     {defines HPFOPEN itemnum 11 }
   density_option            =24;     {defines HPFOPEN itemnum 24 }
   device_class_option       =42;     {defines HPFOPEN itemnum 42 }

type
   pac80  = packed array [1..80] of char;

var
   tfile_num        : integer; {required HPFOPEN filenum parameter  }
   status           : integer; {returns info if error/warning occurs}

   file_name        : pac80;            {declares HPFOPEN itemnum 2  }
   permanent        : integer;          {declares HPFOPEN itemnum 3  }
   update only      : integer;          {declares HPFOPEN itemnum 11 }
   device_class     : pac80             {declares HPFOPEN itemnum 24 }
   density          : integer;          {declares HPFOPEN itemnum 42 }

begin
   tfile_num      :=0;
   status         :=0;
   file_name      :='&tapefile&';   {delimiter is "&"              }
   permanent      :=1;              {search system file domain     }
   update_only    :=5;              {preserves existing data       }
   density        :=1600;           {select this tape density      }
   device_class   :='&tape&';       {system-configured device class name }

   HPFOPEN (tfile_num, status,
     formal_designator_option, file_name,  {formaldesignator option}
     domain_option, permanent,             {domain option}
     access_type_option, update_only       {access type option}
     density_option, density,              {density option}
     device_class_option, device_class     {device class option}
     );

   if status <> 0 then handle_file_error (tfile_num, status);
end;
```

If the HPFOPEN call is successful, a positive integer value is returned in tfile_num, and status returns a value of zero. The file is now open and can be accessed with file system intrinsics. If an error or warning condition is encountered by HPFOPEN, status returns a nonzero value, thus invoking the error-handling procedure handle_file_error.

In appendix A, "HP Pascal/iX Program Examples," Example A-1 uses a similar procedure to open an unlabeled magnetic tape file. For more information about HPFOPEN parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Opening a labeled magnetic tape file

Example 5-6 is an HP Pascal/iX code segment containing an HPFOPEN intrinsic call that

opens a labeled magnetic tape file `labltape`. Use of the HPFOPEN *labeled tape label option* indicates to the file system that the file is opened as a labeled magnetic tape file.

Example 5-6. Opening a Labeled Magnetic Tape File

```
procedure open_labeled_magnetic_tape_file;

const
   formal_designator_option = 2;     {defines HPFOPEN itemnum 2   }
   domain_option            = 3;     {defines HPFOPEN itemnum 3   }
   tape_label_option        = 8;     {defines HPFOPEN itemnum 8   }
   tape_expiration_option   =31;     {defines HPFOPEN itemnum 31  }
   device_class_option      =42;     {defines HPFOPEN itemnum 42  }

type
   pac80  = packed array [1..80] of char;

var
   file_num          : integer; {required HPFOPEN filenum parameter   }
   status            : integer; {returns info if error/warning occurs     }

   file_name         : pac80;             {declares HPFOPEN itemnum 2   }
   old               : integer;           {declares HPFOPEN itemnum 3   }
   tape_label        : pac80;             {declares HPFOPEN itemnum 8   }
   expire_date       : pac80              {declares HPFOPEN itemnum 31  }
   device_class      : pac80              {declares HPFOPEN itemnum 42  }

begin
   file_num        :=0;
   status          :=0;
   file_name       :='&labltape&';    {delimiter is "&"                 }
   old             :=3;               {equivalent to specifying permanent}
   tape_label      :='&tape01&';      {ANSI tape label                  }
   expire_date     :='&05/20/87&'     {when data is no longer useful    }
   device_class    :='&tape&';        {system-configured device name    }

   HPFOPEN (file_num, status,
     formal_designator_option, file_name,    {formaldesignator option}
     domain_option, old,                     {domain option}
     tape_label_option, tape_label,          {labeled tape label option}
     tape_expiration_option, expire_date,    {labeled tape expiration option}
     device_class_option, device_class       {device class option}
           );

   if status <> 0 then handle_file_error (file_num, status);
end;
```

If the HPFOPEN call is successful, a positive integer value is returned in `file_num` and `status` returns a value of zero. The magnetic tape files is now open and ready to be accessed. If an error or warning condition is encountered by HPFOPEN, status returns a nonzero value, thus invoking the error-handling procedure `handle_file_error`.

In appendix A "HP Pascal/iX Program Examples," Example A-2 uses a similar procedure to open a labeled magnetic tape file. For more information about HPFOPEN parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

6 Closing a File

Once your program is finished accessing a file, the program can terminate access to the file with the FCLOSE intrinsic. This chapter discusses various ways that you can use the FCLOSE intrinsic to close disk files and device files. Examples are provided to illustrate important features available through the FCLOSE intrinsic. The following subjects are discussed in detail:

- how the file system closes a file
- closing a disk file
- closing a magnetic tape file

How the File System Closes a File

You terminate access to a file from your program with the FCLOSE intrinsic. The FCLOSE intrinsic applies to both disk and device files. FCLOSE also deallocates the device on which the file resides; however, if your program has several concurrent HPFOPEN/FOPEN calls issued to the same file, the device is not deallocated until the last "nested" FCLOSE intrinsic is executed.

You can use the FCLOSE intrinsic to specify (or change) the disposition of a disk or a magnetic tape file when it is closed. The disposition of a disk or magnetic tape file can be new, temporary, or permanent. If you do not change the disposition of a new file when it is closed, the file and its contents are deleted from the system when the file is closed using FCLOSE.

You can change the disposition of a new file to be either temporary or permanent. A file closed with a temporary disposition is closed as a temporary file. It is deleted from the system when your job/session is terminated. A file closed with a permanent disposition is closed and saved as a permanent file. It remains in the system domain after your job/session ends, and until you purge it.

When you close a file with either a temporary or permanent disposition, MPE/iX conducts a search:

- If the file is to be closed as a temporary file, the job file domain is searched.
- If the file is to be closed as a permanent file, the system file domain is searched.

You are not allowed to have duplicate file names in the same domain. If MPE/iX finds a file of the same name in the searched directory, the file is not closed, and the FCLOSE intrinsic returns an error condition.

You can specify the disposition of a file when it is opened when you use the *final disposition option* or the *file equation option* of the HPFOPEN intrinsic, or the FILE command. Both HPFOPEN options provides the same choices as the *disposition* parameter of FCLOSE, except that you can change the disposition of a file when the file is opened (as opposed to when the file is closed). For more information about HPFOPEN options, refer to the *MPE/iX Intrinsics Reference Manual*.

NOTE Even though you are allowed to specify a file's final disposition when the file is opened, MPE/iX does not search the appropriate directory until you attempt to close that file.

If a conflict occurs between the dispositions specified at file-open time and file-close time, the disposition specification that has the lower positive-integer value takes precedence. For example, if a disposition of temporary (*final disposition option* = 2) is specified by HPFOPEN, and a disposition of permanent (*disposition* = 1) is specified by FCLOSE , the disposition specified by FCLOSE takes precedence. Likewise, if there are conflicts between the disposition specifications of multiple FCLOSE calls on the same file, the disposition specification that has the lower positive-integer value takes precedence when the file is finally closed.

If your program does not issue an FCLOSE intrinsic call on files that have been opened, MPE/iX closes all files automatically when the program's process terminates. In this case, MPE/iX closes all opened files with the same disposition they had before being opened. New files are deleted; old files are saved and assigned to the domain in which they belonged previously, either permanent or temporary; however, if you specified the file's disposition when you opened it with HPFOPEN, that disposition takes effect.

Closing a Disk File

The following examples show how you use the FCLOSE intrinsic to close a disk file:

- "Closing a New Disk File as Permanent" shows an example of an FCLOSE call that closes the file opened in Example 5-1.
- "Closing a Permanent Disk File" shows an example of an FCLOSE call that closes the file opened in Example 5-2.

Closing a new disk file as permanent

Example 6-1 is an HP Pascal/XL code segment containing an HPFOPEN call that opens a new file, and an FCLOSE intrinsic call that changes the disposition of the file to permanent prior to closing it. (Refer to Example 5-1 for details on this HPFOPEN call.)

In Example 6-1, there is a disposition conflict between the FCLOSE call and the HPFOPEN call that opened the file identified by file_num:

- The *disposition* parameter of FCLOSE specifies that the file is to be closed as a permanent file.
- The *final disposition option* of the HPFOPEN call specifies that the file should be closed as a temporary file.

The *disposition* parameter of FCLOSE takes precedence over the final disposition option of HPFOPEN because the integer value of FCLOSE's *disposition* (1) is a smaller positive value than that of HPFOPEN's *final disposition option* (2).

Example 6-1. Closing a New Disk File as Permanent

```
.
.
.
save_temp := 2;
HPFOPEN(file_num, status,
   formal_designator_option, file_name,  {HPFOPEN formaldesignator option}
   record_size_option, line_len,         {HPFOPEN record size option}
   final_disp_option, save_temp,         {HPFOPEN final disp option }
   ASCII_binary_option, ascii            {HPFOPEN ASCII/binary option}
       );
.
.
.
error         := 1;
disposition   := 1;    {close file as a permanent file           }
security_code := 0;    {No additional restrictions               }

FCLOSE ( file_num,     {file_num returned by HPFOPEN             }
         disposition,  {close file with permanent disposition    }
         security_code {no additional restrictions are added     }
       );

if ccode = error then handle_file_error (file_num, 0)
.
```

```
.
.
```

If the file could not be closed because an incorrect `file_num` was specified, or another file of the same name and disposition already exists, `ccode` returns a value of one, thus invoking the error-handling procedure `handle_file_error`.

In Appendix A, "Pascal/XL Program Examples," Example A-1 uses a similar procedure to close a new disk file. For more information about `FCLOSE` parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Closing a permanent disk file

Example 6-2 closes the permanent file opened in Example 5-2. (Refer to Example 5-2 for details on this `HPFOPEN` call.) The disposition of the file is not changed when it is closed. The file remains a permanent disk file.

Example 6-2. Closing a Permanent Disk File

```
.
.
.
HPFOPEN(file_num, status,
   formal_designator_option, file_name,  {HPFOPEN formaldesignator option}
   domain_option, permanent,             {HPFOPEN domain option     }
   access_type_option, update,           {HPFOPEN access type option}
   dynamic_locking_option, lockable,     {HPFOPEN dynamic locking option}
   exclusive_option, shared              {HPFOPEN exclusive option }
   ASCII_binary_option, ascii            {HPFOPEN ASCII/binary option}
       );
.
.
.
error         := 1;
disposition   := 0;    {no change to disposition                 }
security_code := 0;    {No additional restrictions               }

FCLOSE ( file_num,     {file_num returned by HPFOPEN             }
         disposition,  {don't change prior disposition           }
         security_code {no additional restrictions are added     }
       );

if ccode = error then handle_file_error (file_num, 0)
.
.
.
```

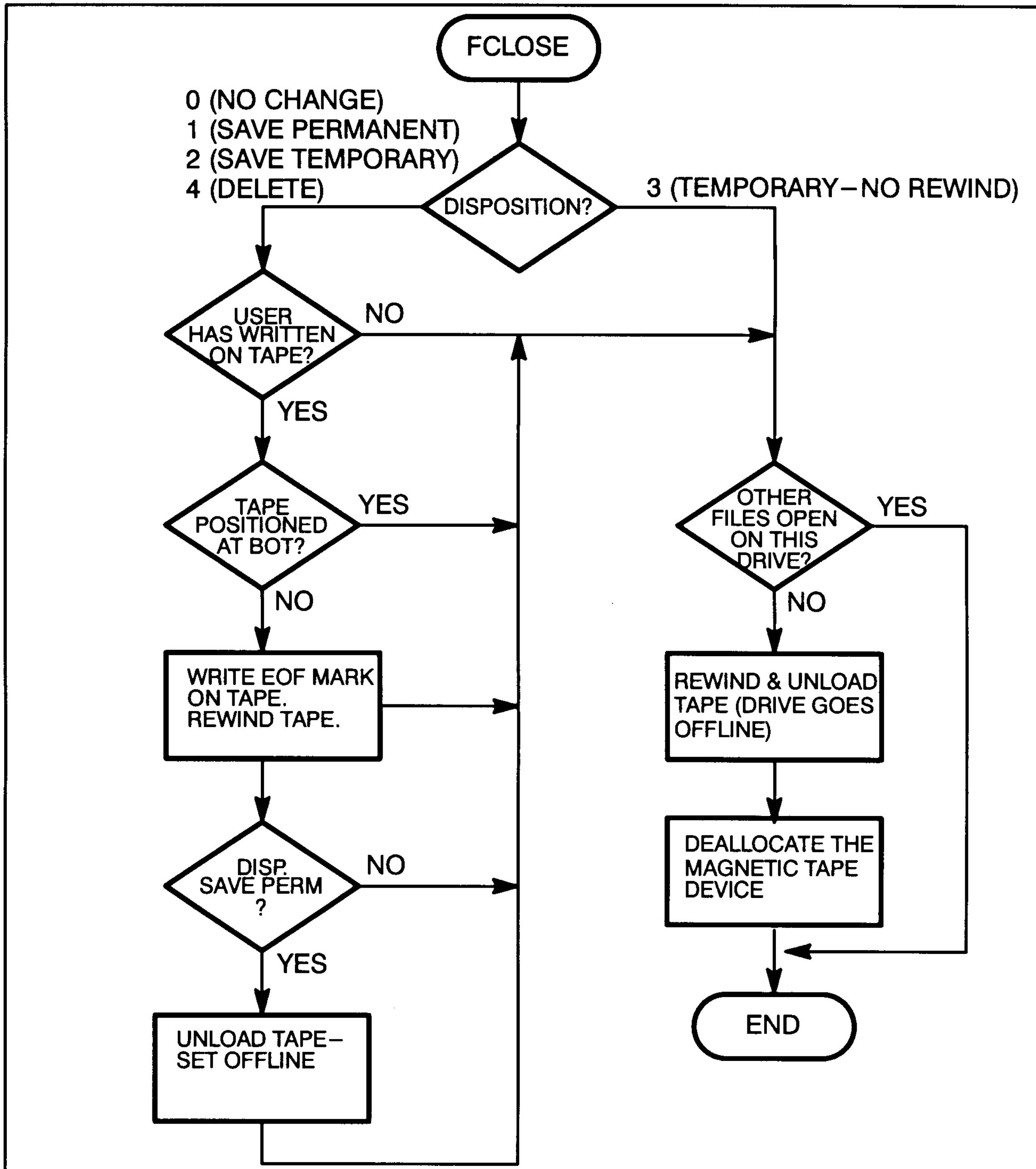
If the file could not be closed because an incorrect `file_num` was specified, or another file of the same name and disposition already exists, `ccode` returns a value of one, thus invoking the error-handling procedure `handle_file_error`.

In Appendix A, "Pascal/XL Program Examples," Example A-5 uses a similar procedure to close a permanent disk file. For more information about `FCLOSE` parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

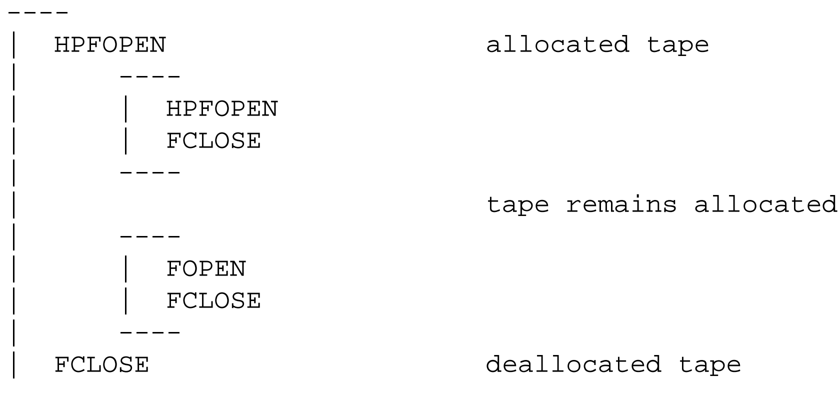
Closing a Magnetic Tape File

The operation of the FCLOSE intrinsic as used with unlabeled magnetic tape is outlined in the flowchart of Figure 6-1..

Figure 6-1. Using the FCLOSE Intrinsic with Unlabeled Magnetic Tape



Note that a tape closed with the temporary no-rewind disposition is rewound and unloaded if certain additional conditions are not met. It is possible for a single process to open a magnetic tape device using the *device class option* of HPFOPEN/FOPEN, then again open the already-allocated device by specifying its logical device number (ldev) using the *device name option* of HPFOPEN/FOPEN. This may be done in such a manner that both magnetic tape files are open concurrently. The second HPFOPEN/FOPEN does not require any operator intervention to allocate the device. When file open and file close calls are arranged in a nested fashion, tape files may be closed without deallocating the physical device, as follows:



Such nesting of HPFOPEN/FOPEN and FCLOSE pairs is required to keep from rewinding a tape closed with FCLOSE. A tape closed with the temporary, no-rewind disposition is rewound and unloaded unless the process closing it has another file currently open on the device.

When a temporary no-rewind tape is deallocated, the file system has not placed an EOF (end-of-file mark) at the end of the data file.

The FCLOSE intrinsic can be used to maintain position when creating or reading a labeled tape file that is part of a volume set. If you close the file with a disposition code of 0 or 3, the tape does not rewind, but remains positioned at the next file. If you close the file with a disposition code of 2, the tape rewinds to the beginning of the file, but is not unloaded. A subsequent request to open the file does not reposition the tape if the sequence (*seq*) subparameter is NEXT or default (1). A disposition code of 1 (save permanent) implies the close of an entire tape volume set.

7 Record Selection and Data Transfer

The chief activities of the file system involve the transfer of data. In this chapter we will examine how this is accomplished. As you read this chapter, keep these considerations in mind:

- How are records selected for transfer?
- What intrinsics are used for data transfer?
- How is the record pointer affected by intrinsics?

The last section of this chapter discusses the major points presented in this chapter as they pertain to magnetic tape files.

Record Pointers

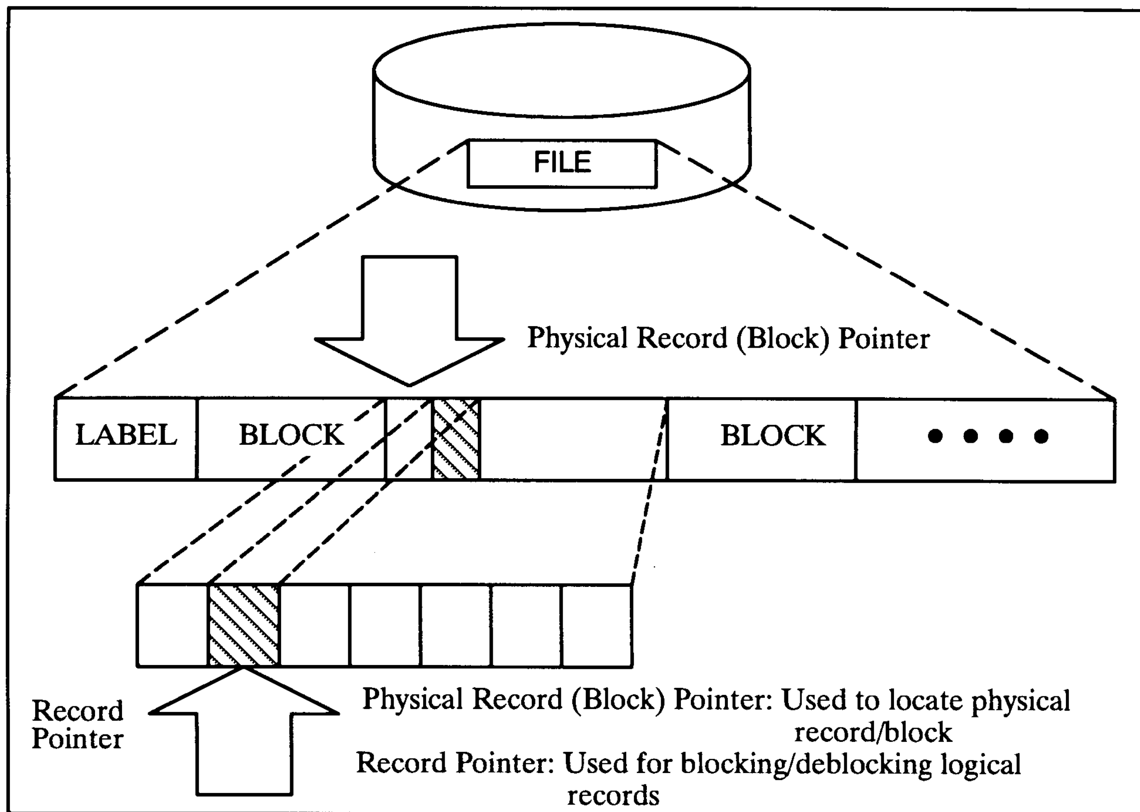
The file system uses record pointers to find specific records for your use. Physical record pointers (also referred to as block pointers) are used to locate specific blocks on disk; logical record pointers blocks and deblocks the logical records in a physical record and indicate specific logical records within a file. A file opened with the *inhibit buffering option* parameter set to BUF (the default) is accessed with a logical record pointer. A file opened with the *inhibit buffering option* parameter set to NOBUF is accessed with a physical record (or block) pointer.

Figure 7-1. shows how the physical and logical record pointers operate together to locate any record in a file. For any record, the physical record pointer indicates the correct block, and the logical record pointer locates the logical record within the block.

The file system uses both the physical and the logical record pointers to locate records. Future references to "record pointer" in this manual will imply this combination.

When you open your file the HPFOPEN/FOPEN intrinsic sets the record pointer to record 0 (the first record in your file) for all operations. If you have opened the file with APPEND access, though (using the *access type option* parameter available in HPFOPEN/FOPEN), MPE/iX moves the record pointer to the end of the file prior to a write operation; this ensures that any data that you write to the file is added to the end of the file rather than written over existing data.

Figure 7-1. Record Pointers



Following initialization, the record pointer may remain in position at the head of your file, or it may be moved by the intrinsics used in record selection.

Record Selection

How are records selected for transfer? The file system intrinsics listed in Table 7-1. are designed to move records to and from your file, but how do they choose the records they want? The record pointer indicates the specific location where a file will be accessed; records can be transferred to or from this location, or the pointer can be moved to another place in the file that you wish to access.

There are five methods of record selection that you can use to access your file:

- sequential access, in which you transfer data to and from the place the record pointer currently indicates
- random access, in which you move the record pointer before transferring data
- update access, in which you choose a record and write a new record over it

- RIO access, in which you access only records that are not deactivated.
- Mapped access is a special type of access available only through the HPFOPEN intrinsic, in which you bypass file system data transfer mechanisms by referencing the file as an array using a pointer declared in your program. Mapped files are discussed in chapter 11, "Accessing a File Using Mapped Access".

Sequential access

When you use this method of record selection, you assume that the record pointer is already where you want it. You transfer your data using the FREAD or FWRITE intrinsic, and the record pointer is automatically set to the beginning of the next record at the end of each read or write; for this reason, this method is also called serial record selection. For fixed-length and undefined-length record files, the file system updates the record pointer by adding the uniform record length to the pointer after you read or write a record; for variable-length record files, the file system takes the byte count from the record being transferred plus some bytes required for overhead, and adds that to the record pointer, resulting in a pointer to the next record.

Random access

If the record pointer is not indicating the location that you want, you can use the random access method to move the pointer and begin your transfer wherever that you like; for this reason, this method is also called controlled record selection.

It is possible to access specific records in a disk file with the FREADDIR and FWRITEDIR intrinsics. The record number to be read or written is specified as one of the parameters in the FREADDIR or FWRITEDIR intrinsic call. Following the read or write operation, the record pointer is set to the next record, as in the default case. Note that FREADDIR and FWRITEDIR may be issued only for a disk file composed of fixed-length or undefined-length records.

Update access

To update a logical record of a disk file, you use the FUPDATE intrinsic. The FUPDATE intrinsic affects the last logical record (or block for NOBUF files) accessed by any intrinsic call for the file named and writes information from a buffer in the stack into this record. Following the update operation, the record pointer is set to indicate the next record position.

The record number need not be supplied in the FUPDATE intrinsic call; FUPDATE automatically updates the last record referenced in any intrinsic call. Note that the file system assumes the record to be updated has just been accessed in some way.

You must open the file containing the record to be updated with the *access type option* parameter of the HPFOPEN/FOPEN call set to update access. In addition, the file must not contain variable-length records.

Table 7-1. Intrinsics for Data Transfer

FREAD	Used for sequential read. May be used with fixed-length, variable-length, or undefined-length record files.

Table 7-1. Intrinsics for Data Transfer

	File must be opened with Read, Read/Write, or Update access. Successful read returns CCE condition code and transfer length; file error results in CCL condition code; end-of-file results in CCG condition code and returns a transfer length of zero.
`FWRITE`	Used for sequential write. May be used with fixed-length, variable-length, or undefined-length record files. File must be opened with Write, Write/SAVE, APPEND, Read/Write or Update access. Successful write returns CCE condition code; file error results in CCL condition code; end-of-file results in CCG condition code.
`FREADDIR`	Used for random-access read. Used only with fixed or undefined-length record files. File must be opened with Read, Read/Write, or Update access. Successful read returns CCE condition code; file error results in CCL condition code; end-of-file results in CCG condition code. No transfer length is returned because you get the amount requested unless an error occurs.
`FREADSEEK`	Used for anticipatory random-access read into file system buffers. Used only with buffered fixed-length or undefined-length record files. File must be opened with Read, Read/Write, or Update access. Successful read returns CCE condition code; file error results in CCL condition code; end-of-file results in CCG condition code.
`FWRITEDIR`	Used for direct write. Use only with fixed-length or undefined-length record files. File must be opened with Write, Write/SAVE, Read/Write or Update access; APPEND not allowed. Successful write returns CCE condition code; file error results in CCL condition code; end-of-file results in CCG condition code.
`FUPDATE`	Used to update previous record (logical or physical). Used only with fixed-length or undefined-length record files.

Table 7-1. Intrinsics for Data Transfer

	File must be opened with Update access. No multirecord update allowed. Successful update returns CCE condition code; file error results in CCL condition code; end-of-file results in CCG condition code.

RIO access

RIO is an access method that permits individual file records to be deactivated. These inactive records retain their relative position within the file. RIO access is intended for use primarily by COBOL programs; however, you can access these files by programs written in any language. You create an RIO file using the *file type option* parameter of HPFOPEN/FOPEN.

RIO files may be accessed in two ways, RIO access and non-RIO access. RIO access ignores the inactive records when the file is read sequentially using the FREAD intrinsic, and these records are transparent to you; however, they can be read by random access using FREADDIR. They may be overwritten both sequentially and randomly using FWRITE, FWRITEDIR, or FUPDATE. With non-RIO access, the internal structure of RIO blocks is transparent.

Multiple Record Transfers

In almost all applications, programs conduct input/output in normal recording mode, where each read or write request transfers one logical record to or from the data stack. In certain cases, however, you may want your program to read or write, in a single operation, data that exceeds the logical record length defined for the input or output file.

For instance, you may want to read four 128-byte logical records from a file to your data stack in a single 512-byte data transfer. Such cases usually arise in specialized applications. Suppose, for example, that your program must read input from a disk file containing 256-byte records. This data, however, is organized as units of information that may range up to 1024 bytes long; in other words, the data units are not confined to record boundaries. Your program is to read these units and map them to an output file, also containing 256-byte records.

You can bypass the normal record-by-record input/output, instead receiving data transfers of 1024 bytes each, by specifying multirecord mode (MR) using the multirecord option parameter in your HPFOPEN/FOPEN call or FILE command. For example:

```
:FILE BIGCHUNK; REC=-256,1,U;NOBUF;MR
                                    \
                                     Specifies multirecord mode
```

The essential effect of multirecord mode is to make it possible to transfer more than one block in a single read or write. This mode effectively ignores block boundaries, and permits transfers of as much data as you wish; it does not, however, break up blocks. Your transfers must begin on block boundaries. In order to take advantage of multirecord mode, you must

also set the the *inhibit buffering option* parameter to NOBUF in your HPFOPEN/FOPEN call or FILE command.

When you read from a file in multirecord mode, you may not read beyond the EOF (end-of-file marker). When you write to a file in multirecord mode, you may write only up to the block containing the file limit. If your transfer exceeds its limit, a condition code of CCG is returned, data is transferred only up to the limit, and the FREAD intrinsic returns a transfer length of 0.

NOTE To obtain the actual transfer length for your data use the FCHECK intrinsic, as described in the *MPE/iX Intrinsics Reference Manual*. The transfer length is returned in the TLOG parameter of FCHECK.

Control Operations

There may be times when you want to move the record pointer to a particular place without necessarily transferring any data. There are three general categories for this type of record selection:

Spacing: Move the record pointer backward or forward.

Pointing: Reset the record pointer.

Rewinding: Reset the pointer to record 0.

Spacing

To space forward or backward in your file, use the FSPACE intrinsic. Its syntax is

```
FSPACE(filenum,displacement);
```

The displacement parameter gives the number of records to space from the current record pointer. Use a positive number for spacing forward in the file or a negative number for spacing backward.

You can use the FSPACE intrinsic only with files that contain fixed-length or undefined-length records; variable-length record files are not allowed. The FSPACE intrinsic may not be used when you have opened your file with APPEND access; the file system returns a CCL condition if you attempt to use it in this case. Spacing beyond the EOF results in a CCG condition, and the record pointer is not changed.

Pointing

To request a specific location for the record pointer to indicate, use the FPOINT intrinsic. Its syntax is

```
FPOINT(filenum,recnum);
```

Use the recnum parameter to specify the new location for the record pointer; recnum is the

record number relative to the start of the file (record 0).

You can use The FPOINT intrinsic only with files that contain fixed-length or undefined-length records; variable-length record files are not allowed. The FPOINT intrinsic may not be used when you have opened your file with Append access; the file system returns a CCL condition if you attempt to use it in this case. It is legal to point to any record between the start of file and the file limit. Subsequent reads will fail if the data pointer is positioned beyond the EOF. If the data pointer is positioned beyond the EOF and a subsequent write is done, this will become the new EOF and all data between the old and new EOF will be initialized with the fill character.

Rewinding

When you "rewind" your file, you set the record pointer to indicate record 0, the first record in your file. Use the FCONTROL intrinsic with a control code of 5 to accomplish this. FCONTROL's syntax in this case would be

```
FCONTROL(filenum,5,dummyparam);
```

Issuing this intrinsic call sets the record pointer to record 0. You can use FCONTROL with fixed-length, variable-length, or undefined-length record files; you can use it with any access method.

NOTE	FCONTROL's control code 5 has a special meaning when used with Append access. The file system sets the record pointer to record 0, as with other access modes, but at the time of the next write operation to the file, the record pointer is set to the end of the file so that no data is overwritten.

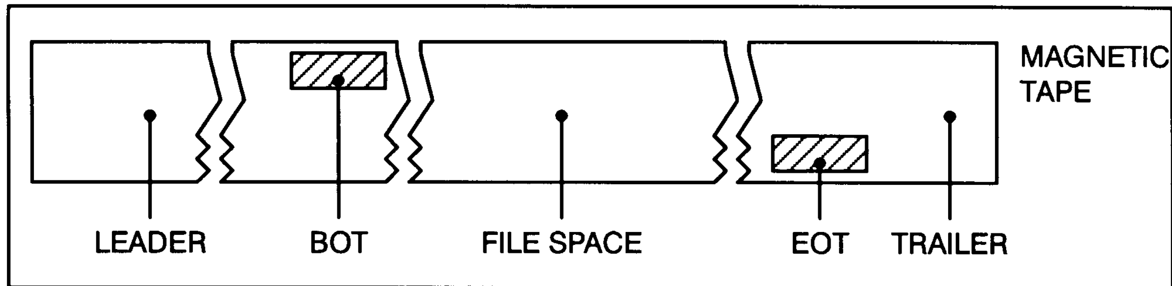
For more information about the FSPACE, FPOINT, and FCONTROL intrinsics, consult the *MPE/iX Intrinsics Reference Manual*.

Magnetic Tape Considerations

The most common medium for storage of a device file is magnetic tape. This section describes the matters that you should keep in mind when you work with your magnetic tape files.

Every standard reel of magnetic tape designed for digital computer use has two reflective markers located on the back side of the tape (opposite the recording surface). As illustrated in Figure 7-2., one of these marks is located behind the tape leader at the beginning-of-tape (BOT) position, and the other is located in front of the tape trailer at the end-of-tape (EOT) position. These markers are sensed by the tape drive itself and their position on the tape (left or right side) determines whether they indicate the BOT or EOT positions:

Figure 7-2. Magnetic Tape Markers



As far as the magnetic tape hardware and software are concerned, the BOT marker is much more significant than the EOT marker because BOT signals the start of recorded information; but EOT simply indicates that the remaining tape supply is running low, and the program writing the tape should bring the operation to an orderly conclusion. The difference in treatment of these two physical tape markers is reflected by the file system intrinsics when the file being read, written, or controlled is a magnetic tape device file. The following paragraphs discuss the characteristics of each appropriate intrinsic.

FWRITE. If the magnetic tape is unlabeled (as specified in the `HPFOPEN`/`FOPEN` intrinsic or `FILE` command) and a user program attempts to write over or beyond the physical EOT marker, the `FWRITE` intrinsic returns an error condition code (CCL). The actual data is written to the tape, and a call to `FCHECK` reveals a file error indicating end-of-tape. All writes to the tape after the EOT marker has been crossed transfer the data successfully, but return a CCL condition code until the tape crosses the EOT marker again in the reverse direction (rewind or backspace).

If the magnetic tape is labeled (as specified in the `HPFOPEN`/`FOPEN` intrinsic or `FILE` command), a CCL condition code is not returned when the tape passes the EOT marker. Attempts to write to the tape after the EOT marker is encountered cause end-of-volume (EOV) labels to be written. A message then is printed on the operator's console requesting another reel of tape to be mounted.

FREAD. A user program can read data written over an EOT marker and beyond the marker into the tape trailer. The intrinsic returns no error condition code (CCL or CCG) and does not initiate a file system error code when the EOT marker is encountered.

DDS tape drives do not permit an `FREAD` past the marker. With DDS drives, every `FWRITE` updates the EOT marker and does not permit a rewrite of previous data.

FSPACE. A user program can space records over or beyond the EOT marker without receiving an error condition code (CCL or CCG) or a file system error. The intrinsic does, however, return a CCG condition code when a logical file mark is encountered. If the user program attempts to backspace records over the BOT marker, the intrinsic returns a CCG condition code and remains positioned on the BOT marker.

FCONTROL (Write EOF). If a user program writes a logical end-of-file (EOF) mark on a magnetic tape over the reflective EOT marker, or in the tape trailer after the marker, hardware status is saved to return `END-OF-TAPE` on the next `FWRITE`. The file mark is actually written to the tape.

FCONTROL (FORWARD SPACE TO FILE MARK). A user program that spaces forward to logical tape marks (EOFs) with the FCONTROL intrinsic cannot detect passing the physical EOT marker. No special condition code is returned.

FCONTROL (BACKWARD SPACE TO FILE MARK). The EOT reflective marker is not detected by FCONTROL during backspace file (EOF) operations. If the intrinsic discovers a BOT marker before it finds a logical EOF, it returns a condition code of CCE and treats the BOT as if it were a logical EOF. Subsequent backspace file operations requested when the file is at BOT are treated as errors and return a CCL condition code and set a file system error to indicate INVALID OPERATION.

In summary, except for FCONTROL, only those intrinsics that cause the magnetic tape to write information are capable of sensing the physical EOT marker. If a program designed to read a magnetic tape needed to detect the EOT marker, it could be done by using the FCONTROL intrinsic to read the physical status of the tape drive itself. When the drive passes the EOT marker and is moving in the forward direction, tape status bit 5 (%2000) is set and remains on until the drive detects the EOT marker during a rewind or backspace operation. Under normal circumstances, however, it is not necessary to check for EOT during read operations. The responsibility for detecting end-of-tape and concluding tape operations in an orderly manner belongs to the program that originally created (wrote) the tape.

A program that needed to create a multiple reel tape file would normally write tape records until the status returned from FWRITE indicated an EOT condition. Writing could be continued in a limited manner to reach a logical point at which to break the file. Then several file marks and a trailing tape label would typically be added, the tape rewound, another reel mounted, and the data transfer continued. The program designed to read such a multitape file must expect to find and check for the EOF and label sequence written by the tape's creator. Since the logical end of the tape may be somewhat past the physical EOT marker, the format and conventions used to create the tape are of more importance than determining the location of the EOT.

8 Writing to a File

This chapter describes, through program examples, various ways that you can use file system intrinsics to transfer data from your program to a disk file or a device file. This chapter is intended to illustrate topics introduced in chapter 7, "Record Selection and Data Transfer". This chapter is divided into the following topics:

- "Sequential Access and Random Access" discusses two common methods of record selection and data transfer, and uses examples to illustrate the use of the `FWRITE` and `FWRITEDIR` intrinsics.
- "Writing to $STDLIST" discusses how your program can use the `PRINT` and `FWRITE` intrinsics to write data from your program to the job/session standard list device, `$STDLIST`. An example illustrates the use of the `PRINT` intrinsic to write a character string to `$STDLIST`.
- "Writing Messages to the system console" discusses how your program can send messages from your program to the system console, and request a reply from the operator. Examples illustrate the use of the `PRINTOP` and `PRINTOPREPLY` intrinsics.
- "Writing to a Magnetic Tape File" discusses how you can write data to two kinds of magnetic tape files: unlabeled tape files and labeled tape files. Examples illustrate the use of the `FWRITE` intrinsic to write data to both types of tape files.
- "Writing a File Label to a Labeled Tape File" provides an example of using the `FWRITELABEL` intrinsic to write a user-defined file label to a labeled magnetic tape file.

Sequential Access and Random Access

Two of the most frequently used methods of transferring data to a file from your program are sequential access and random access.

When you use sequential access to write data to a file, you write data to the record currently pointed to by the record pointer. You use the `FWRITE` intrinsic to write data sequentially to a disk file or device file. When you open a file with any form of write access (except Append) specified in the *access type option* of `HPFOPEN`/`FOPEN`, the file is opened with the record pointer set to the first record in the file. When you have accomplished the write operation, the file system automatically sets the record pointer to point to the beginning of the next record in the file. Both disk files and device files can be accessed with the `FWRITE` intrinsic.

When you use random access to write data to a disk file, you write data to any record in the file by specifying where you want the file system to set the record pointer prior to the write

operation. You use the FWRITEDIR intrinsic to write data randomly to a disk file. You must specify in FWRITEDIR which record that you want to write to. The file system sets the record pointer to the selected record, then transfers the data to the record from your program's stack. When you have accomplished the write operation, the file system automatically sets the record pointer to point to the beginning of the next record in the file. Only disk files can be accessed with the FWRITEDIR intrinsic.

The following examples illustrate the use of file system intrinsics to perform sequential access writes and random access writes to a disk file.

Writing to a disk file using sequential access

Example 8-1 is an HP Pascal/iX code segment that copies logical records sequentially from an unlabeled tape file (indicated by variable tape_file_num) and uses FWRITE to write them to a disk file (indicated by variable disk_file_num). The operation is performed in a loop. The loop ends when the FREAD intrinsic encounters an EOF marker on the tape (indicating the end of the file).

Example 8-1. Writing to a Disk File Using Sequential Access

```
procedure copy_tape_file_to_disk_file;

var
   record          : packed array [1..80] of char; {declare record        }
   end_of_file     : boolean;                      {declare exit condition}
   record_length   : shortint;                     {size of record read   }
   length          : shortint;                     {declare parameter     }
   control_code    : 0..65535;                     {declare parameter     }

begin
   end_of_file  := false;                   {initialize exit condition    }
   control_code := 0;                       {initialize to default        }
   length       : -80;                      {size of record to be copied  }

   repeat                                   {loop until exit condition    }

      record_length := FREAD  (tape_file_num, record, length);
      if ccode = ccl then                 {check condition code for error }
        handle_file_error  (tape_file,  3)
      else
        if ccode = ccg then               {FREAD returns ccg if EOF       }
        end_of_file := true               {exit condition encounter encountered}
        else
          begin
            FWRITE( disk_file_num,        {identity returned by HPFOPEN   }
                 record,                  {read from tape_file_num        }
                    record length,        {actual size of record          }
                    control_code          {default                        }
                  );
           if ccode <> cce then         {check condition code for error }
              handle_file_error (disk_file, 5);
         end
    until end_of_file;
end                                       {end procedure                  }
```

If an error is encountered by either FWRITE or FREAD, the condition code CCL is returned to the program, thus invoking the procedure handle_file_error. For more information about FWRITE parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about using the FREAD intrinsic, refer to chapter 9, "Reading from a File". For more information about opening a file, refer to chapter 5, "Opening a File". In appendix A, "HP Pascal/iX Program Examples," example A-1 uses a similar procedure to copy records from a tape file to a disk file.

Writing to a disk file using random access

Example 8-2 is an HP Pascal/iX code segment that reads records sequentially from old_disk_file and writes them into new_disk_file. Assume that both files have been opened already with calls to HPFOPEN/FOPEN. The end-of-file (EOF) using the FWRITEDIR of old_disk_file is determined with the FGETINFO intrinsic and assigned to the variable record_num.

Example 8-2. Writing to a Disk File Using Random Access.

```
procedure copy_from_old_file_to_new_file;

var
   record_num     : integer;
   buffer         : packed array [256] of char;
   end_of_file    : boolean;
   read_length    : integer;
   length         : shortint;

begin
   end_of_file  := false;                   {initialize exit condition    }
   record_num   := 0;                       {initialize record pointer    }
   length       := 128                      {also means 256 bytes         }

   FGETINFO (old disk_file,,,,,,,,,,rec);    {locate the EOF in old_disk_file}

      if ccode = ccl then
        handle_file_error  (old_disk_file);   {error check on intrinsic call}

      repeat                                {Copy the records in the reverse}
                                            {orders from old disk file      }

                                            {to the new disk file           }

     read_length := FREAD (old_disk_file, buffer, length);
     if ccode = ccl then
        handle_file_error (old_disk_file)
        else
        if ccode = ccg then                {check for exit condition}
           end_of_file := true
        else begin
              rec := rec - 1               {decrement record pointer}
              FWRITEDIR(new_disk_file, buffer, read_length, record_num);
              if ccode <> cce then
                   handle_file_error (new_discfile);  {error check    }
           end
      until end_of_file                   {exit loop if exit condition true}
```

```
end;                                          {end procedure            }
```

The operation is performed in a loop. Before each write operation, `record_num` is decremented. The loop ends when the `FREAD` intrinsic encounters an EOF in `old_disk_file` (indicating the end of the file). For more information about `FWRITEDIR` intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about the `FREAD` intrinsic, refer to chapter 9, "Reading from a File". In appendix A, "HP Pascal/iX Program Examples", example A-3 uses a similar routine to copy records using the random access method of data transfer to write date from one file to another.

Writing to $STDLIST

You can write data from your program to your program's standard list device `$STDLIST` using two intrinsics:

- `PRINT`
- `FWRITE`

Normally, `$STDLIST` for jobs is a line printer and for sessions a terminal. You can write a string of ASCII characters from an array in your program to `$STDLIST` with the `PRINT` intrinsic. You do not need to use `HPFOPEN/FOPEN` to open the standard list device in order to use `PRINT`.

NOTE The `PRINT` intrinsic is limited in its usefulness in that `FILE` commands are not allowed. In addition, you cannot use the `FCHECK` intrinsic to determine error conditions encountered by `PRINT`. You may find it more convenient (and a better programming practice) to use the `HPFOPEN/FOPEN` intrinsic to open the file `$STDLIST`, then write to this file using `FWRITE`.

You can also use the `FWRITE` intrinsic to write data from your program to the standard list device `$STDLIST`, if you opened `$STDLIST` with `HPFOPEN/FOPEN`. In this case, the `HPFOPEN/FOPEN` call returns a file number that identifies `$STDLIST`. You would then write to `$STDLIST` sequentially using `FWRITE`. For more information about opening `$STDLIST`, refer to chapter 5, "Opening a File".

Example 8-3 is an HP Pascal/iX code segment that contains a `PRINT` intrinsic call that transmits a message to `$STDLIST`.

Example 8-3. Writing to $STDLIST Using PRINT

```
              .
              .
              .
   var
      message         : packed array [1..72] of char;  {declare PRINT parm}
      message_length  : shortint;                      {declare PRINT parm}
      controlcode     : 0..65535;                      {declare PRINT parm}
              .
```

```
         .
         .
   message        := 'WRITING A MESSAGE TO THE STANDARD LIST DEVICE.';
   message_length := -46              {message is 46 bytes long      }
   control_code   := 0;
   PRINT ( message,                   {message written to $STDLIST   }
           message_length,            {number of bytes in message    }
           controlcode                {set to default                }
         );
         .
         .
         .
```

For more information about PRINT parameters, refer to the *MPE/iX Intrinsics Reference Manual*. In appendix A, "HP Pascal Program Examples," example A-2 uses the PRINT intrinsic to write messages to $STDLIST.

Writing Messages to the System Console

Two intrinsics are available that allow you to print a character string directly from your program to the system console:

- PRINTOP transmits an ASCII character string from your program to the system console.
- PRINTOPREPLY transmits an ASCII character string from your program to the system console, and solicits a reply from the system operator.

Writing a message to the system console

Example 8-4 is an HP Pascal/iX program segment that illustrates how your program can call the PRINTOP intrinsic to transmit a message from a character array in your program to the System Console.

Example 8-4. Writing a Message to the System Console

```
         .
         .
         .
var
   message       : packed array [1..56] of char;  {declare PRINTOP parm}
   length        : shortint;                      {declare PRINTOP parm}
   controlcode   : 0..65535;                      {declare PRINTOP parm}
         .
         .
         .
   message       := 'Message to Operator';      {message to transmit   }
   length        := -19               {actual length in bytes          }
   controlcode   := 0;                {set to default                  }
   PRINTOP ( message,
             length,
             controlcode
            );
         .
```

```
          .
          .
```

The PRINTOP intrinsic transmits a maximum of 56 ASCII characters to the system console. Longer messages are truncated to 56 characters. For more information about PRINTOP intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Writing a message to the system console and requesting a reply

The PRINTOPREPLY intrinsic can be used to transmit a message from an array in your program to the system console and to request that a reply be returned. The message that you send must be no longer than 50 characters in length. PRINTOPREPLY can return a maximum of 31 ASCII characters to your program. For example, a program could ask the system operator if the line printer contains a certain type of form. If the response is affirmative, the program could then write information on these forms.

Example 8-5 is an HP Pascal/iX code segment containing a PRINTOPREPLY intrinsic call. The program is asking the system operator if the line printer device LP contains the correct forms. The program is requesting that the system operator respond with a simply YES or NO response. The program takes appropriate action based upon the characters returned in reply.

Example 8-5. Writing a Message to the System Console and Requesting a Reply

```
          .
          .
          .
   var
      message      : packed array [1..50] of char;  {PRINTOREPLY parameter}
      length       : shortint;                      {PRINTOREPLY parameter}
      zero         : shortint;                      {PRINTOREPLY parameter}
      reply        : packed array [1..31] of char;  {PRINTOREPLY parameter}
      expected_length: shortint;                    {PRINTOREPLY parameter}
      counter
          .
          .
          .
      message      := 'Does device LP contain the correct forms? [Y/N]';
      length       := -47              {length of message              }
      zero         := 0;
      reply        := '   ';           {initialize reply               }
      expected_length := -3            {expected reply Y/YES/N/NO      }
      PRINTOREPLY ( message,           {message sent to system console }
                    length,            {length of message in range 0..50 }
                    zero,              {required, but not used. Set to 0 }
                    reply              {reply returned in this array   }
                    expected_length    {length of reply in range 0..31 }
                  );
          .
          .
          .
```

The actual length of the System Operator's reply is returned to expected_length. For more information about PRINTOREPLY intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Writing to a Magnetic Tape File

The following discussion pertains to writing data to two different types of magnetic tape files.

- unlabeled magnetic tape files
- labeled magnetic tape files

Unless you specifically create and open a labeled magnetic tape file, the file system opens an unlabeled magnetic tape file when you specify a tape drive using either the *device name option* or *device class option* of HPFOPEN/FOPEN. For more information about opening both unlabeled and labeled magnetic tape files, refer to chapter 5, "Opening a File".

When you are writing records to an unlabeled magnetic tape file, you must take into consideration characteristics of magnetic tape that do not apply to files on other devices. For example, if a user program attempts to write over or beyond the physical EOT marker, the FWRITE intrinsic returns an error condition code (CCL). The actual data is written to the tape, and a call to FCHECK reveals a file error indicating END-OF-TAPE. All writes to the tape after the EOT tape marker has been crossed transfer the data successfully, but return a CCL condition code until the tape crosses the EOT marker again in the reverse direction (rewind or backspace). For more information about magnetic tape considerations, refer to chapter 7, "Record Selection and Data Transfer".

Writing records to a labeled tape file differs slightly from writing to an unlabeled tape file. If the magnetic tape is unlabeled and a user program attempts to write over or beyond the physical EOT marker, the FWRITE intrinsic returns an error condition code (CCL). The actual data has been written to the tape, and a call to FCHECK reveals a file error indicating END-OF-TAPE. All writes to the tape after the EOT tape marker has been crossed transfer the data successfully, but return a CCL condition code until the tape crosses the EOT marker again in the reverse direction (rewind or backward).

If the magnetic tape is labeled, a CCL condition code is not returned when the tape passes the EOT marker. Attempts to write to the tape after the EOT marker is encountered cause end-of-volume (EOV) labels to be written. A message then is printed on the operator's console requesting another reel of tape to be mounted.

The following headings provide examples of file system intrinsic calls that illustrate:

- writing to an unlabeled magnetic tape file
- writing to a labeled magnetic tape file
- writing a user-defined file label on a labeled tape file

Writing to an unlabeled magnetic tape file

Example 8-6 is an HP Pascal/iX code segment that writes user-supplied data to the unlabeled magnetic tape file opened in example 5-5. For information about the HPFOPEN call that returns the file number in the variable unlabeled_tape_file, refer to example 5-5.

Example 8-6. Writing to an Unlabeled Magnetic Tape File

```
          .
          .
          .
   var
      control_code : 0..65535;        {Declare FWRITE parm.             }
      record_length: shortint;        {Declare FWRITE parm              }
      file_record  : record_type;     {Record to be written to file;    }
                                      {record_type is 256-byte          }
                                      {fixed-length record.             }
          .
          .
          .
      record_length:= -256;           {Number of bytes in record.       }
      control_code := 0;              {Default specified                }
      FWRITE ( unlabeled_tape_file,   {HPFOPEN returned file number.    }
               file_record            {Record to be passed              }
               record_length          {Size of file_record.             }
               control_code           {Required, but ignored.           }
             );

   if ccode = CCL                     {check FWRITE condition code      }
      then handle_file_error (labeled_tape_file);
          .
          .
          .
```

If the `FWRITE` intrinsic encounters an error condition (CCL), an error-handling procedure, `handle_file_error`, is invoked. `FWRITE` returns a CCG condition code if the EOF is reached. For details concerning `FWRITE` intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

Writing to a labeled magnetic tape file

Example 8-7 is an HP Pascal/iX code segment that writes user-supplied data to the labeled magnetic tape file opened in Example 5-6. For information about the `HPFOPEN` call that returns the file number in the variable `labeled_tape_file`, refer to example 5-6.

Example 8-7. Writing to a Labeled Magnetic Tape File

```
           .
           .
           .
   var
      control_code : 0..65535;        {Declare FWRITE parm.             }
      record_length: shortint;        {Declare FWRITE parm              }
      file_record  : record_type;     {Record to be written to file;    }
                                      {record_type is 256-byte          }
                                      {fixed-length record.             }
           .
           .
           .
      record_length:= -256;           {Number of bytes in record.       }
      control_code := 0;              {Default specified                }
      FWRITE ( labeled_tape_file,     {HPFOPEN returned file number.    }
               file_record            {Record to be passed              }
```

```
              record_length          {Size of file_record.              }
              control_code           {Required, but ignored.            }
            );

   if ccode = CCL                    {check FWRITE condition code       }
      then handle_file_error (labeled_tape_file);
             .
             .
             .
```

If the FWRITE intrinsic encounters an error condition (CCL), an error handling procedure handle_file_error is invoked. FWRITE returns a CCG condition code if the EOF is reached. For more information about FWRITE intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about opening files, refer to chapter 5, "Opening a File".

Writing a File Label to a Labeled Tape File

User-defined labels are used to further identify files and may be used in addition to the ANSI-standard labels. User-defined labels are written on files with the FWRITELABEL intrinsic instead of with the HPFOPEN/FOPEN intrinsic, as is the case for writing ANSI-standard labels.

User-defined labels for labeled tape files differ slightly from user-defined labels for disk files, in that user-defined labels for tape files must be 80 bytes (40 half-words) in length. The tape label information need not occupy all 80 bytes, however, and you can set unused portions of the space equal to blanks.

In order to write a user-defined header label, the FWRITELABEL intrinsic must be called before the first FWRITE to the file. MPE/iX does, however, write user-defined trailer labels if FWRITELABEL is called after the first FWRITE.

NOTE User-defined labels may not be written on unlabeled magnetic tape files.

Example 8-8 is an HP Pascal/iX code segment that writes a user-label to the labeled magnetic tape file opened in example 5-6. For information about the HPFOPEN call that returns the file number in the variable LABELED_TAPE_FILE, refer to example 5-6.

Example 8-8. Writing a User-Label to a Labeled Magnetic Tape File.

```
          .
          .
          .
   var
      counter      : integer;        {Initialize counter                }
      label_length : shortint;       {FWRITELABEL length parm           }
      user_label   : packed array [1..80] of char;
          .
          .
          .
      label_length := 40;            {40 half-words required length     }
```

```
      for counter  := 1 to 80 do       {Loop to fill array with            }
         user_label [counter] := ' '; {ASCII blanks.                       }
      user_label   := 'tape01 user header label no. 1'; {Overwrite first }
                                            {30 bytes with label name }

   FWRITELABEL (labeled_tape_file,     {Required parameter                 }
                user_label,            {Required parameter                 }
                label_length           {Optional parameter                 }
               );

   if ccode = CCL or CCG               {check FWRITELABEL condition code   }
      then handle_file_error (labeled_tape_file);
               .
               .
               .
```

If ccode indicates that the FWRITELABEL intrinsic encountered an error condition (either CCL or CCG), an error handling procedure handle_file_error is invoked. For more information about FWRITELABEL intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about opening files, refer to chapter 5, "Opening a File".

Writing User Data in ANSI Labels

It is possible to write data into bytes 5/21 of the HDR1 record of an ANSI tape label. In all, 17 bytes are available. If you write more than 8 bytes into the record, the 9th byte (Byte 13) must be a period (".").

For example, to write the string "FRANKSTN COUNCIL" into bytes 5/21, mount your tape and then do this:

```
FILE FRANKSTN.COUNCIL;DEV=TAPE;REC=-80,,F,ASCII;LABEL=BUDGET,ANS
FCOPY FROM=datafile;TO=*FRANKSTN.COUNCIL
```

where *datafile* is the name of a disk file. This coerces the string "FRANKSTN.COUNCIL" into bytes 5/21 of the HDR1 record, and it places "BUDGET" into the VOL1 record of the tape.

The "file" and "group" names are right-justified.

5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
F	R	A	N	K	S	T	N	.		C	O	U	N	C	I	L

In order to retrieve the string recorded in bytes 5/21, you must create a program that uses the intrinsics FOPEN (or HPFOPEN) and FFILEINFO. The program must do two things:

- It must FOPEN (or HPFOPEN) the tape device.
- It must employ FFILEINFO with option 45 to retrieve the "file" identifier.

The tape containing such a label must be mounted before you run the program.

9 Reading from a File

This chapter describes, through program examples, various ways that you can use file system intrinsics to transfer data to your program from a disk file or device file. This chapter is intended to illustrate topics introduced in chapter 7, "Record Selection and Data Transfer". This chapter is divided into the following topics:

- "Sequential Access and Random Access" discusses two common methods of record selection and data transfer, and uses examples to illustrate the use of the `FREAD` and `FREADDIR` intrinsics.
- "Reading from $STDIN" discusses how your program can use the `READ`, `READX`, and `FREAD` intrinsics to read data from the job/session standard input device, `$STDIN`. An example illustrates the use of the READ intrinsic to read a character string from `$STDIN`.
- "Reading from a Magnetic Tape File" provides an example of using the `FREAD` intrinsic to read data from file located on magnetic tape.
- "Reading a File Label from a Labeled Tape File" provides an example of using the `FREADLABEL` intrinsic to read a user-defined file label located on a labeled magnetic tape file.

Sequential Access and Random Access

Two of the most frequently used methods of transferring data from a file to your program are sequential access and random access.

When you use sequential access to read data from a file, you read data from the record currently pointed to by the record pointer. You use the FREAD intrinsic to read data sequentially from a disk file or device file. When you open a file with any form of Read access specified in the *access type option* of HPFOPEN/FOPEN, the file is opened with the record pointer set to the first record in the file. When you have accomplished the read operation, the file system automatically sets the record pointer to point to the beginning of the next record in the file. Both disk files and device files can be accessed with the FREAD intrinsic. When you use random access to read data from a disk file, you read data from any record in the file by specifying where you want the file system to set the record pointer prior to the read operation. You use the FREADDIR intrinsic to randomly access records in a disk file. You must specify in FREADDIR which record that you want to read from. The file system sets the record pointer to the selected record, then transfers the data from the record to your program's stack. When you have accomplished the read operation, the file system automatically sets the record pointer to point to the beginning of the next record in the file. Only disk files can be accessed with the FREADDIR intrinsic

The following examples illustrate the use of file system intrinsics to perform sequential access reads and random access reads from a disk file.

Reading from a disk file using sequential access

Example 9-1 is an HP Pascal/iX code segment that uses the FREAD intrinsic to read records sequentially from a disk file. Example 9-1 contains a loop construct, where records are read sequentially from disk_file and written to the file new_file (both files opened elsewhere by HPFOPEN/FOPEN calls). The files are both standard ASCII files with fixed-record format, each record 256 bytes in length. When a logical end-of-file (EOF) is reached, a condition code of CCG is returned by FREAD. The loop ends when FREAD encounters the EOF and returns the CCG condition to the program.

Example 9-1. Reading from a Disk File Using Sequential Access

```
              .
              .
              .
   var
      expected_length: shortint;                    {required by FREAD   }
      record         : packed array [1..256] of char; {declare record type }
      control_code   : 0..65535;                    {required by FWRITE  }
      record_length  : shortint;                    {expected record length}
      end_of_file    : boolean;                     {declare exit condition}
              .
              .
              .
   begin
```

```
      end_of_file := false;                     {initialize exit condition    }
      expected_length := -256;                  {record size of file          }
      control_code := 0                         {set to default               }

   repeat                                                 {begin loop         }
      record_length := FREAD  (  disk_file,     {file number from HPFOPEN     }
                                 record,        {data transferred to here     }
                                 expected_length {record size of file         }
                              );
      if ccode = ccl then handle_file_error (disk_file)  {error check        }
      else if ccode = ccg then end_of_file :=  true      {exit condition check}
           else begin
              FWRITE( new_file,                 {file number from HPFOPEN     }
                      record,                   {data transferred here        }
                      record_length,            {value returned by FREAD      }
                      control_code              {required; set to default     }
                    );
           end
   until end_of_file;             {loop ends when exit condition encounted   }
                  .
                  .
                  .
```

If an error is encountered by either `FREAD` or `FWRITE`, the condition code CCL is returned to the program, thus invoking the procedure `handle_file_error`. For more information about `FREAD` parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about using the `FWRITE` intrinsic, refer to chapter 8, "Writing to a File". For more information about opening a file, refer to chapter 5, "Opening a File".

Reading from a disk file using random access

Example 9-2 is an HP Pascal/iX code segment that, within a loop construct, calls the `FREADDIR` intrinsic to read a record whose record number has been selected by the procedure `select_record` and returned in the variable `record_number`. The example then prints the selected record to the standard list device `$STDLIST` using the `PRINT` intrinsic.

Example 9-2. Reading from a Disk File Using Random Access

```
                  .
                  .
                  .
var
   record          : packed array [1..30] of char; {declare record type    }
   record_length   : shortint;                  {expected record length    }
   read_length     : shortint;                  {actual bytes read by FREAD}
   record_number   : integer;                   {which record to read      }
   control_code    : shortint;                  {required by PRINT         }
   end_of_file     : boolean;                   {declare exit condition    }
                  .
                  .
                  .
   control_code := 0;                           {default condition         }
```

```
record_length := -30;                        {file record size 30 bytes }
record_number := 0;                          {initialize variable       }
end_of_file := false;                        {initialize exit condition }

repeat                                       {begin loop                }
   select_record (record_number);
   read_length := FREADDIR (data_file,    {HPFOPEN file number        }
                            record,       {record read from data_file }
                            record_length,{expected length of record  }
                            record_number {returned from select_record }
                           );
   if ccode = ccl then handle_file_error (data_file)   {error check    }
   else if ccode = ccg then end_of_file := true   {check for exit condition}
         else begin
                 PRINT  (record,              {returned by FREADDIR    }
                         read_length,         {returned by FREADDIR    }
                         control_code         {set to default condition}
                        );
                if ccode <> cce then handle_file_error  (data_file)
             end
until end_of_file;                           {exit if exit condition true }
         .
         .
         .
```

Assume that a disk file identified by `data_file` has been opened elsewhere by an `HPFOPEN`/`FOPEN` call. Also, assume that procedure `select_record` prompts the user for a valid record number of a record in `data_file`. The loop is repeated until the `FREADDIR` intrinsic encounters an end-of-file condition, or an error condition is returned by an intrinsic.

If an error is encountered by either `FREADDIR` or `PRINT`, procedure `handle_file_error` is invoked. For more information about `FREADDIR` parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about using the `PRINT` intrinsic, refer to chapter 8, "Writing to a File". For more information about opening a file, refer to chapter 5, "Opening a File".

Increasing I/O performance using FREADSEEK

If you know in advance that a certain record is to be read from a file with the `FREADDIR` intrinsic, you can speed up the I/O process by issuing an `FREADSEEK` intrinsic call.

The `FREADSEEK` intrinsic moves the record from disk to virtual memory. Then, when the `FREADDIR` intrinsic call is issued, the record is transferred from virtual memory to the buffer in the stack specified by `FREADDIR` without having to perform I/O. The use of `FREADSEEK` enhances the I/O process, because the `FREADDIR` call does not make the file system perform a physical I/O.

Reading From $STDIN

You can read data from your program's standard input device ($STDIN) by using one of the following intrinsics:

- READ
- READX
- FREAD

The job/session input device is the source of all MPE/iX commands relating to a job or session and is the primary source of all ASCII information input to the job or session. You can read a string of ASCII characters from the job/session input device into an array in your program with the READ and READX intrinsics. The READ and READX intrinsics are identical, except that the READX intrinsic reads input from $STDINX instead of $STDIN. The $STDINX file is equivalent to $STDIN, except that records with a colon (:) in the first column of a line indicate the end-of-file to $STDIN, and only the commands :EOD, and EOF indicate the end of file for $STDINX.

NOTE The READ and READX intrinsics are limited in their usefulness in that FILE commands are not allowed. In addition, you cannot use the FCHECK intrinsic to determine error conditions encountered by READ or READX. You may find it more convenient (and a better programming practice) to use the HPFOPEN/FOPEN intrinsic to open the files $STDIN or $STDINX, then issue FREAD calls against these files.

If the standard input device ($STDIN) and the standard list device ($STDLIST) are opened with an HPFOPEN/FOPEN intrinsic call, the FREAD and FWRITE intrinsics can be used with these devices. For example, the FREAD intrinsic can be used to transfer information entered from a terminal to a buffer in the stack, and the FREAD intrinsic can be used to transfer information from your stack directly to the standard list device.

Example 9-3 is an HP Pascal/iX code segment that uses the PRINT intrinsic to prompt a user for a file designator, then uses the READ intrinsic to read the input from $STDIN. Assume that the file designator is then returned to a procedure that calls HPFOPEN to open a file with the *formaldesignator option* passing the file name specified by the user.

Example 9-3. Reading from $STDIN Using READ

```
procedure get_file_designator (var file_name : packed array [1..80] of
char);

var
   message         : packed array [..80] of char;  {holds prompt to user }
   length          : shortint;                     {length of prompt      }
   control_code    : shortint;                     {required by PRINT     }
   read_length     : shortint;                     {length read by READ   }
   expected_length : shortint;                     {size of message array}

begin
```

```
      massage :='Please input a valid file reference'; {specify prompt      }
      length := -35;                                   {length of prompt    }
      control_code := 0                                {default condition   }
      expected_length := -80
      PRINT (message,
             length,
             control_code
            );
      if ccode <> cce then handle_file_error;
      else begin
            read_length := READ ( file_name,          {read data to output parm}
                                  expected_length     {length of file_name     }
                        );
            if ccode <> cce then handle_file_error;
        end
   end;
```

If an error is encountered by either `READ` or `PRINT`, procedure `handle_file_error` is invoked. For more information about `READ` parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about using the `PRINT` intrinsic, refer to chapter 8, "Writing to a File". For more information about opening a file, refer to chapter 5, "Opening a File". For more information about file designators, refer to chapter 3, "Specifying a File Designator". In appendix A, "HP Pascal/iX Program Examples", example A-2 uses a routine similar to example 9-3 to prompt the user for a valid file reference.

Reading From a Magnetic Tape File

Example 9-4 is an HP Pascal/iX code segment that reads records sequentially from an unlabeled magnetic tape file (indicated by variable `tape_file_num`) and uses `FWRITE` to write them to a disk file (indicated by variable `disk_file_num`). The operation is performed in a loop. The loop ends when the `FREAD` intrinsic encounters an EOF marker on the tape (indicating the end of the tape file).

Example 9-4. Reading From a Magnetic Tape File

```
procedure copy_tape_to_disk_file;


var
   record          : packed array [1..30] of char; {declare record        }
   end_of_file     : boolean;                      {declare exit condition }
   record_length   : shortint;                     {size of record read    }
   length          : shortint;                     {declare parameter      }
   control_code    : 0..65535;                     {declare parameter      }

begin
   end_of_file := false;                          {initialize exit condition}
   control_code := 0;                             {initialize to default    }
   length := -80;                                 {size of record to be copied}

   repeat                                         {loop until exit condition}
```

```
      record_length := FREAD (tape_file_num, record, length);
      if ccode = ccl then
         handle_file_error (tape_file, 3)
      else
         if ccode = ccg then                {FREAD returns ccg if EOF     }
         end_of_file := true                {exit condition encountered   }
         else
           begin
              FWRITE( disk_file_num,        {identity returned by HPFOPEN }
                      record,               {read from tape_file_num      }
                      record_length,        {actual size of record        }
                      control_code          {default                      }
                    );
              if ccode <> cce then         {check condition code for error}
              handle_file_error (disk_file, 5);
           end

      entil end_of_file;
end;                                                  {end procedure      }
```

If an error is encountered by either `FREAD` or `FWRITE`, procedure `handle_file_error` is invoked. For more information about `FREAD` intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about the `FWRITE` intrinsic, refer to chapter 8, "Writing to a File". In appendix A, "HP Pascal/iX Program Examples", example A-1 uses a similar procedure to copy records from a tape file to a disk file.

Reading a File Label from a Labeled Tape File

The `FREADLABEL` intrinsic is used to read a user-defined label located on a labeled magnetic tape file or a labeled disk file. To read a user-defined header label, the `FREADLABEL` intrinsic must be called before the first `FREAD` is issued for the file. Execution of the first `FREAD` causes MPE/iX to skip past any unread user-defined header labels.

Example 9-5 is an HP Pascal/iX code segment that reads a user label located in the user-defined label portion of `data_file`, a labeled magnetic tape file. The label is printed to `` `STDLIST `` (identified by `list_file`) using the `FWRITE` intrinsic. Assume that both `data_file` and `list_file` have been opened elsewhere with calls to `HPFOPEN/FOPEN`.

Example 9-5. Reading a User Label from a Labeled Magnetic Tape File

```
   procedure read_user_label;

   var
      label          : packed array [1..80] of char;  {holds label from file }
      length         : shortint;                      {length of label       }
      control_code   : 0..65535;                      {required by FWRITE    }

   begin
      length := 40                                    {required ANSI label size}
      control_code := 0                               {set to default        }
```

```
      FREADLABEL ( data_file,                          {file number of tape file}
                   label,                              {returns label          }
                   length                              {# of halfwords to read}
                 );
      if ccode <> cce then handle_file_error (data_file);
      FWRITE  ( list_file,                        {output to $STDLIST      }
                label,                            {label read by FREADLABEL }
                length,                           {length of label          }
                control_code                      {default condition        }
              );
      if ccode <> cce then handle_file_error (list_file);
   end;                                                {end read_user_label   }
```

If an error is encountered by either FREADLABEL or FWRITE, procedure handle_file_error is invoked. For more information about FREADLABEL intrinsic parameters, refer to the *MPE/iX Intrinsics Reference Manual*. For more information about the FWRITE intrinsic, refer to chapter 8, "Writing to a File". In appendix A, "HP Pascal/iX Program Examples", example A-2 uses a similar procedure to read a user label from a labeled magnetic tape file.

10 Updating a File

You can use the FUPDATE intrinsic to update a logical record of a disk file. FUPDATE affects the last logical record (or block for NOBUF files) accessed by any intrinsic call for the file named, and writes information from a buffer in the stack into this record. Following the update operation, the record pointer is set to indicate the next record position. The record number need not be supplied in the FUPDATE intrinsic call; FUPDATE automatically updates the last record referenced in any intrinsic call. Note that the file system assumes that the record to be updated has just been accessed in some way.

The disk file containing the record to be updated must have been opened with the *access type option* parameter of HPFOPEN/FOPEN set to update access. In addition, the file must not contain a variable-length record format. Example 10-1 is an HP Pascal/iX code segment that illustrates how to use the FUPDATE intrinsic to update records in a disk file being shared by multiple concurrent accessors. The program segment also uses file system locking intrinsics (FLOCK and FUNLOCK) to guarantee exclusive access to the file while the update occurs.

The pertinent code from Example 10-1 is shown below:

```
.
.
.
read_length := FREAD (disk_file_num, inbuf, 128);
.
.
.
FUPDATE (disk_file_num, inbuf, 128);
.
.
.
```

The statements above are in a loop that follows this algorithm:

1. Read the record from the file identified by disk_file_num using the FREAD intrinsic.
2. Write the record to STDLIST to be reviewed by user.
3. Read new data input to STDIN by user and modify record in program.
4. Using FUPDATE intrinsic, write the updated record to the location in disk_file_num indicated by last intrinsic call (in this case, the FREAD call shown above).

Example 10-1. Updating a Disk File

```
procedure update_disk_file;

{***********************************************************}
```

```
{ procedure update_disk_file updates records in the disk file  }
{ with the replacement records read from $STDIN                 }
{**************************************************************}

var
   dummy       : integer;
   inbuf       : array [1..80] of char;
   end_of_file : boolean;
   read_length : integer;

begin
                                {Lock the file and suspend      }
   end_of_file  := false;
   FLOCK (disk_file_num, 1);
   if ccode = ccl then
     handle_file_error (disk_file_num, 0);
                                {Begin loop                     }
repeat
  { Read record from disk file, then write record to $STDLIST; }
  { read updated record number from $STDIN and update           }
  { the disk file with the input record and unlock disk file.  }

   read_length := FREAD (disk_file_num, inbuf, 128);
                                                {read in record}
   if ccode = ccg then
       end_of_file := true              {exit condition         }
   else
       begin
        FWRITE (std_list, inbuf, -20, octal('320'));
                                           {user reviews record}
        if ccode <> cce then
           handle_file_error (std_list, 5);
        dummy := FREAD (std_in, inbuf[20], 5);
                                           {input updated field}
        if ccode = ccl then
           handle_file_error (std_list, 6);
        else
            if ccode = ccg then
              end_of_file := true;              {exit condition}

FUPDATE (disk_file_num, inbuf, 128);     {update record }
        if ccode <> cce then
           handle_file_error (disk_file_num, 7);
        end
   until end_of_file;                           {test for EOF  }
   FUNLOCK (disk_file_num);           {final unlock of disk file}
         if ccode <> cce then
           handle_file_error (disk_file_num, 2);
end:                                 {end update_file           }
```

In appendix A, "HP Pascal Program Examples," example A-5 is an HP Pascal/iX program that uses the procedure in example 10-1 to update records in a disk file. For more information about `FUPDATE` parameters, refer to the *MPE/iX Intrinsics Reference Manual*.

NOTE A magnetic tape device is not designed to enable the update/replacement of a single record in an existing file. Problems occurs in maintaining the integrity of tape records if you attempt to perform a record update operation directly to a magnetic tape file. You should update individual records of a magnetic tape file only when you are copying the entire contents of that file to another file.

---

11 Accessing a File Using Mapped Access

A major enhancement to the MPE/iX file system is **mapped file access**, a method that allows you to access a file directly through memory load and store instructions. Mapped file access is available through three HPFOPEN intrinsic optional parameters:

- Item #18 *short-mapped option* returns a 32-bit value of type address.
- Item #21 *long-mapped option* returns a 64-bit value of type address.
- Item #87 large-mapped option returns a 64-bit value of type address. Long-mapped and large-mapped options are identical with the exception that attempts to open a file larger than 4GB - 64KB with the long-mapped option will result in a failure. The large-mapped option may be used to open any file and is therefore considered more reliable.

How to Access a File Mapped

You can access a file mapped by declaring a short (32-bit) or long (64-bit) pointer variable within a program and passing that variable to the appropriate `HPFOPEN` option. The `HPFOPEN` intrinsic returns the variable pointing to the beginning of the data area of the opened file.

After `HPFOPEN` returns the address of the file, you simply reference the pointer as an array. The machine architecture ensures integrity and protection of the file.

The following file types are allowed any type of access (Read, Write, Read/Write, and so forth) when opened using mapped access options:

- standard disk files with fixed-length or undefined-length record formats

The following file types are allowed read-only access when opened using mapped access options:

- standard disk files with variable-length record formats
- KSAM files opened with *copy mode option* enabled

The following file types are not allowed to be opened using mapped access options:

- relative I/O (RIO) files
- message (MSG) files
- circular (CIR) files
- device files

You can use all applicable file system intrinsics with files opened for mapped access, including all data transfer intrinsics; however, when mixing data transfer intrinsic calls (such as `FREAD` and `FWRITE`) with mapped access, you must take into consideration the data type (ASCII/binary) of the file, the record format, and the record size. Otherwise, data written to the file using mapped access may not make sense when read by `FREAD`.

When you open a file using mapped access and write data to that file, you must use the `FPOINT` and `FCONTROL` intrinsics to reset the EOF before you close the file. Otherwise, all data you write to the file after the EOF will be lost when you close the file. In the case of a newly created file, the EOF initially points to record zero.

NOTE When you access a file with mapped access you are bypassing file system services that set various file system pointers automatically, including the EOF and the logical record pointer. You are responsible for resetting the EOF prior to closing a file that you have accessed mapped. Also, file system posting is bypassed; so, if data recovery is needed you should use `FCONTROL` *controlcodes* 2 and/or 6 to post data and update the EOF periodically. Heavy use of the `FCONTROL` intrinsic to post data and set the EOF degrades performance due to the overhead of the extra posting.

When attempting to set the EOF to the file limit for a file opened with mapped access, the

FCLOSE intrinsic disposition parameter, bit 10:1 set to 1 will force the EOF to the file limit. This overcomes the problem of trying to FPOINT to the file limit.

Advantages of mapped access

Mapped access to a file can be much faster than access through normal file system intrinsics. This is especially the case when you are accessing a smaller file randomly rather than sequentially. When accessing a file mapped, there is no file system overhead associated with a specific reference to the file. The only difference between accessing a file mapped and accessing normal memory is the locality of the access and the protection strategy.

NOTE It is possible to show a degradation of performance if an application that accesses files sequentially is modified to access those files mapped. Normal system reads prefetch multiple records per read. Mapped file access has no method of prefetching the data, consequently, some performance penalty is paid by additional overhead on page faults.

There are two perspectives you can take on mapped file access:

1. A file is accessible as virtual memory. The advantages from this perspective are high performance and fast response time from the file system.
2. Virtual memory is accessed through the file system. The advantages from this perspective are
   - Virtual memory can be easily saved permanently.
   - Virtual memory can be checkpointed.
   - Virtual memory can be easily shared through a common naming convention.

Short-mapped access

Item #18 *short-mapped option* is available in the HPFOPEN intrinsic to provide you with shared virtual memory. A short pointer is returned in an optional *item* parameter. You can use the pointer as a large array of any type to efficiently access the file.

A file created using the *short-mapped option* can be up to four megabytes in size. A process can have open at the same time up to six megabytes of files that are opened using the *short-mapped option*. For larger file needs, a file created using the *long-mapped option* is required.

An error results if you attempt to open a file using the *short-mapped option* that you have previously opened normally or with the *long-mapped option*.

You cannot access a loaded program file or a loaded library file using either mapped access option. In addition, you cannot load a file that is currently being accessed mapped.

Long-mapped access

Item #21 *long-mapped option* is available in the HPFOPEN intrinsic to provide you with

access to large amounts of shared virtual memory. You can use the pointer as a large array of any type to efficiently access the file.

A file created using the *long-mapped option* can be up to 4GB - 64KB in size. There is no practical limit of the number of long-mapped access files that a process can have open at a time.

You cannot access a loaded program file or a loaded library file using either mapped access option. In addition, you cannot load a file that is currently being accessed mapped.

Advantages of long-mapped access over short-mapped access are:

- You can access much larger files than you can using the *short-mapped option*.
- You can open files that were opened previously with any options (as long as the exclusive status of the file is not violated)

The disadvantage of long-mapped access is that it may be slower than short mapped access because of the need to load a space register to access the long-pointer space. long-mapped access can be as much as four times slower than short-mapped access, although long-mapped access still can be faster than accessing the file through the file system data transfer intrinsics.

The degradation can be minimized if you make the references to the long pointer space in a localized part of your code. This way it may be possible for the system to keep the pointer to the file's virtual space loaded into a space register rather than to repeatedly load and unload it.

Large-mapped access

Item #87, large-mapped option is similar to the long-mapped option. However, due to the possible issues of trying to dereference a data structure that straddles a space ID boundary, the large-mapped option is required whenever opening a large file for user mapped access. Specifying this option indicates that the programmer is aware of the issue of cross-SID buffers and has coded the application to correctly handle this situation. In all other respects the large-mapped option is identical to the long-mapped option.

Opening a File Mapped

Example 11-1 illustrates how a file is created and opened with short-mapped access. This HP Pascal/XL program segment opens the file, then writes data to the file with assignments to the array structure. The procedure then sets the EOF and closes the file.

The file is then reopened short-mapped, and data is retrieved before the file is closed and purged.

Example 11-1. Opening a Mapped File

```
procedure Mapped_File_Example;
type
  record_t = record               {** defines an 80-byte record **}
```

```
               a_record : packed array [1..80] of char;
             end;
  file_t = array [1..50000] of record_t;
                                    {** define a 4,000,000 byte array **}
var
  access,domain        : integer;
  dummy                : shortint;
  file_name            : packed array [1..20] of char;
  file_number          : integer;
  file_ptr             :^file_t; {** pointer to the file **}
  filesize             : integer;
  index, rec           : integer;
  create_domain_perm   : integer;
  read_write_access    : integer;
  domain_old           : integer;
  status               : record
                            case integer of
                              0:    (all:    integer);
                              1:    (info:   shortint;
                                     subsys:  shortint);
                          end;
const
  file_name_option     =   2;
  domain_option        =   3;
  filesize option      =  35;
  short_mapped_option  =  18;
  access_type_option   =  11;


begin
                {** initialize item values for the HPFOPEN **}
     file_name            := '%EXAMPLE%';
     create_domain_PERM   := 4;
     domain_OLD           := 3;
     filesize             := 15265;
     read_write_access    := 4;

                 {** create a short-mapped file **}
     HPFOPEN (
               file_number, status,
               file_name_option, file_name
               domain_option,create_domain_PERM,
               filesize option, filesize,
               short_mapped_option, file_ptr,
               access_type_option, read_write_access,
             );
                 {** put some data into the file **}
     for rec := 1 to 100 do
          for index := 1 to 80 do
               file_ptr^[rec].a_record[index] :=
                         Chr (((rec - 1) Mod 26) + 65);

                    {** set the logical record pointer **}
     FPOINT (file_number, 33);
     FCONTROL (file_number, 6, dummy);        {** set the EOF **}
     FCLOSE (file_number, 0, 0);            {**close the file **}
```

```
                    {** re-open the same short-mapped file **}
    HPFOPEN (
              file_number, status,
              file_name_option, file_name,
              domain_option, domain_OLD,
              short_mapped_option, file_ptr,
            );

                    {** retrieve some data you put in file **}
    for rec := 1 to 100 do
    begin
         write ('Record-', rec:4, ' ');
         for index := 1 to 20 do
              write (file_ptr^[rec].a_record[index];
         writeln;
    end;
                              {** close and purge the file **}
    FCLOSE (file_number, 4, 0);
end;
```

New Intrinsics

The following intrinsics have been added to make dealing with user mapped files simpler. The intrinsics provide for basic pointer manipulation, data movement and memory initialization. These intrinsics safely hadle all cross-SID buffer problems, removing the user from having to deal with these situations. These intrinsics can be safely used on long-mapped or large-mapped files.

HPFADDTOPOINTER

NM callable only.

This routine can be used to perform arithmetic on a 64-bit pointer value. Byte offsets can be added to or subtracted from a pointer by specifying eithger a positive or negative *offset* value.

Syntax

```
                       @64      I64        @64       I32
HPFADDTOPOINTER(base_ptr,offset,return_ptr,status,
```

Parameters

base_ptr **64-bit pointer by reference (required)**

The base_ptr can be a 64-bit pointer to an object of any type.

offset **64-bit signed integer by reference (required)**

The offset can be any positive or negative value. Specifying a positive value will move the return_ptr forward from the previous base_ptr, while a negative value will move the return_ptr backward from the base_ptr.

return_ptr **64-bit pointer by reference (required)**

The return_ptr is an output parameter that will have the new pointer value returned to it. It can be a 64-bit pointer to an object of any type

status **32-bit signed integer by reference (optional)**

Returns the status of the HPFADDTOPOINTER call. If no errors or warnings are encountered, status returns 32 bits of zero. If errors or warnings are encountered, status is interpretted as two 16-bit fields.
Bits (0:16) comprise status.info. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16) comprise status.subsys. The value represents the subsystem that set the status information.

Operation Notes

No attemt is made to verify that the pointer value returned is a legitimate pointer to a valid object. Any invalid pointers will be detected and generate errors when the pointers are dereferenced.

Related Information

Manual *Accessing Files Programmer's Guide*

HPFFILLDATA

NM callable only.

This routine can be used to efficiently initialize a buffer with a specified character value.

Syntax

```
                  I64      @64          CV       I32
   HPFFILLDATA(count,buffer_ptr,fill_char,status,
```

Parameters

count **64-bit signed integer by reference (required)**

A positive count of the number of bytes in the buffer indicated by the buffer_ptr parameter that should be initialized.

buffer_ptr **64-bit pointer by value (required)**

A pointer to the buffer that should be initialized. The buffer_ptr may point to any valid object in your stack, heap, or a file that has been opened with user mapped access.

fill_char **Character value by value (required)**

The character value that should be used to initialize the specified buffer. Any value in the range of 0 through 255 can be specified, including all printable and non-printable ASCII characters.

status **32-bit signed integer by reference (optional)**

Returns the status of the HPFFILLDATA call. If no errors or warnings are encountered, status returns 32 bits of zero. If errors or warnings are encountered, *status* is interpretted as two 16-bit fields.
Bits (0:16) comprise status.info. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16) comprise status.subsys. The value represents the subsystem that set the status information.

Operation Notes

None.

Related Information

Manual *Accessing Files Programmer's Guide*

HPFMOVEDATA

NM callable only.

This routine can be used to efficiently move data from a source buffer to a target buffer.

Syntax

```
                I64     @64          @64       I32
   HPFMOVEDATA(count,source_ptr,target_ptr,status,
```

Parameters

count **64-bit signed integer by reference (required)**

The count parameter allows the caller to specify the number of bytes to move from the source buffer to the target buffer.

source_ptr **64-bit pointer by value (required)**

The source_ptr can be a 64-bit pointer to any valid object that the calling process has access to. The buffer may be in the caller's stack, heap, or obtained by opening a file with user mapped access

target_ptr **64-bit pointer by value (required)**

The target_ptr can be a 64-bi pointer to any valid object that the calling proceThis intrinsic is especially useful when the source and target buffers are overlapping. The HPFMOVEDATARTOL intrinsic is typically used when the target buffer's address is to the right (larger) of the source buffer's address. Moving the data from the right to the left ensures that the data in the source buffer is copied to the target buffer before it is overwritten itselfss has access to. The buffer may be in the caller's stack, heap, or obtained by opening a file with user mapped access.

status **32-bit signed integer by reference (optional)**

Returns the status of the HPFMOVEDATA call. If no errors or warnings are encountered, status returns 32 bits of zero. If errors or warnings are encountered, *status* is interpretted as two 16-bit fields.
Bits (0:16) comprise status.info. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16) comprise status.subsys. The value represents the subsystem that set the status information.

Operation Notes

When calling the HPFMOVEDATA intrinsic it is important to ensure that the source and target buffers are not overlapping. The results of a HPFMOVEDATA call when source and target buffers are overlapping are undefined. If source and target buffers are overlapping, the HPFMOVEDATALTOR or HPFMOVEDATARTOL intrinsics should be used.

Related Information

Manual *Accessing Files Programmer's Guide*

HPFMOVEDATALTOR

NM callable only.

This routine can be used to efficiently move data from a source buffer to a target buffer. If the source and target buffers were viewed horizontally, like a line of text, the data movement is performed by starting at leftmost position of the source buffer (to the leftmost position of the target buffer) and proceeding to the rightmost

Syntax

```
                     I64    @64         @64       I32
HPFMOVEDATALTOR(count,source_ptr,target_ptr,status,
```

Parameters

count **64-bit signed integer by reference (required)**

The count parameter allows the caller to specify the number of bytes to move from the source buffer to the target buffer.

source_ptr **64-bit pointer by value (required)**

The source_ptr can be a 64-bit pointer to any valid object that the calling process has access to. The buffer may be in the caller's stack, heap, or obtained by opening a file with user mapped access

target_ptr **64-bit pointer by value (required)**

The target_ptr can be a 64-bi pointer to any valid object that the calling process has access to. The buffer may be in the caller's stack, heap, or obtained by opening a file with user mapped access.

status **32-bit signed integer by reference (optional)**

Returns the status of the HPFMOVEDATALTOR call. If no errors or warnings are encountered, status returns 32 bits of zero. If errors or warnings are encountered, *status* is interpretted as two 16-bit fields.
Bits (0:16) comprise status.info. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16) comprise status.subsys. The value represents the subsystem that set the status information.

Operation Notes

This intrinsic is especially useful when the source and target buffers are overlapping. The HPFMOVEDATALTOR intrinsic is typically used when the target buffer's address is to the left (smaller) of the source buffer's address. Moving the data from the left to the right ensures that the data in the source buffer is copied to the target buffer before it is overwritten itself.

Related Information

Manual *Accessing Files Programmer's Guide*

HPFMOVEDATARTOL

NM callable only.

This routine can be used to efficiently move data from a source buffer to a target buffer. If the source and target buffers were viewed horizontally, like a line of text, the data movement is performed by starting at rightmost position of the source buffer (to the rightmost position of the target buffer) and proceeding to the leftmost

Syntax

```
                    I64     @64         @64      I32
HPFMOVEDATARTOL(count,source_ptr,target_ptr,status,
```

Parameters

count **64-bit signed integer by reference (required)**

The count parameter allows the caller to specify the number of bytes to move from the source buffer to the target buffer.

source_ptr **64-bit pointer by value (required)**

The source_ptr can be a 64-bit pointer to any valid object that the calling process has access to. The buffer may be in the caller's stack, heap, or obtained by opening a file with user mapped access

target_ptr **64-bit pointer by value (required)**

The target_ptr can be a 64-bi pointer to any valid object that the calling process has access to. The buffer may be in the caller's stack, heap, or obtained by opening a file with user mapped access.

status **32-bit signed integer by reference (optional)**

Returns the status of the HPFMOVEDATARTOL call. If no errors or warnings are encountered, status returns 32 bits of zero. If errors or warnings are encountered, *status* is interpretted as two 16-bit fields.
Bits (0:16) comprise status.info. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16) comprise status.subsys. The value represents the subsystem that set the status information.

Operation Notes

This intrinsic is especially useful when the source and target buffers are overlapping. The HPFMOVEDATARTOL intrinsic is typically used when the target buffer’s address is to the right (larger) of the source buffer’s address. Moving the data from the right to the left ensures that the data in the source buffer is copied to the target buffer before it is overwritten itself.

Related Information

Manual *Accessing Files Programmer's Guide*

12 Sharing a File

Accessing and controlling a file that is open only to you is a relatively simple matter. When your file is being accessed by several users simultaneously, each user must be aware of special considerations for this shared file. As you read this chapter, keep these considerations in mind:

- How will others be allowed concurrent access to your file?
- Will the concurrent access need special management?

NOTE In most cases, the following discussions pertain only to non-message files. For more information about using message files; refer to the *Interprocess Communication Programmer's Guide*.

Simultaneous Access of Files

When an HPFOPEN/FOPEN request is issued for a file, that request is regarded as an individual accessor of the file and a unique file number and other file control information is established for that file. Even when the same program issues several different HPFOPEN/FOPEN calls for the same file, each call is treated as a separate accessor. Under the normal (default) security provisions of MPE/iX, when an accessor opens a file not presently in use, the access restrictions that apply to this file for other accessors depend upon the access mode requested by this initial accessor:

- If the first accessor opens the file for Read-only access, any other accessor can open it for any other type of access (such as Write-only or Append), except that other accessors are prohibited Exclusive access.
- If the first accessor opens the file for any other access mode (such as Write-only, Append, or Update), this accessor maintains Exclusive access to the file until it closes the file; no other accessor can access the file in any mode.

Programs can override these defaults by specifying other options in HPFOPEN/FOPEN intrinsic calls. Users running those programs can, in turn, override both the defaults and programmatic options through the FILE command. The options are listed in Table 12-1. on page 154 The actions taken by MPE/iX when these options are in effect, and simultaneous access is attempted by other HPFOPEN/FOPEN calls, are summarized in Figure 12-1. The

action taken depends upon the current use of the file versus the access requested.

Table 12-1. File Sharing Restriction Options

ACCESS RESTRICTION	FILE\ PARAMETER	DESCRIPTION
Exclusive Access	EXC	After file is opened, prohibits concurrent access in any mode through another HPFOPEN/FOPEN request, whether issued by this or another program, until this program issues FCLOSE or terminates.
Exclusive Write Access	SEMI	After file is opened, prohibits concurrent Write access through another HPFOPEN/FOPEN request, whether issued by this or another program, until this program issues FCLOSE or terminates.
Shareable Access	SHR	After file is opened, permits concurrent access to file in any mode through another HPFOPEN/FOPEN request issued by this or another program, in this or any session or job.

Exclusive access

This option is useful when you wish to update a file and wish to prevent other users or programs from reading or writing on the file while you are using it; thus, no user can read information that is about to be changed, nor can he alter that information. To override the programmatic option under which the file would be opened and request exclusive access, you could use the EXC keyword parameter in the FILE command:

```
FILE DATALIST;EXC     <----   Requests exclusive access
RUN FLUPDATE
```

NOTE In all cases, when the first accessor to a file opens it with Exclusive (EXC) access, all other attempts to open the file fails.

Figure 12-1. Requested Access Granted, Unless Noted

		REQUESTED ACCESS GRANTED, UNLESS NOTED					
	Current Use	HPFOPEN/ FOPEN for Input		HPFOPEN/ FOPEN for Output		HPFOPEN/ FOPEN for Input/Output	
Requested Access		SHR/ MULTI/ GMULTI	SEMI	SHR/ MULTI/ GMULTI	SEMI	SHR/ MULTI/ GMULTI	SEMI
HPFOPEN/ FOPEN for Input	SHR	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted
	SEMI	Requested Access Granted	Requested Access Granted	Error Message	Error Message	Error Message	Error Message
HPFOPEN/ FOPEN for Output	SHR	Requested Access Granted	Error Message	Requested Access Granted	Error Message	Requested Access Granted	Error Message
	SEMI	Requested Access Granted	Error Message	Error Message	Error Message	Error Message	Error Message
HPFOPEN/ FOPEN for Input/ Output	SHR	Requested Access Granted	Input Granted	Requested Access Granted	Input Granted	Requested Access Granted	Input Granted
	SEMI	Requested Access Granted	Input Granted	Error Message	Error Message	Error Message	Error Message

Semi-exclusive access

This option allows other accessors to read the file, but prevents them from altering it. When appending new part numbers to a file containing a parts list, for instance, you might use this option to allow other users to read the current part numbers at the same time that you are adding new ones to the end of the file. You could request this option as follows:

```
FILE PARTSLST;SEMI     <----   Requests semi-exclusive access
RUN FLAPPEND
```

Shared access

When opened with the share option, a file can be shared (in all access modes) among several HPFOPEN/FOPEN requests, whether they are issued from the same program, different programs within the same job or session, or programs running under different jobs or sessions. Each accessor transfers its input/output to and from the file with its own unique buffer, using its own set of file control information and specifying its own buffer size and blocking factor. Effectively, each accessor accesses its own copy of that portion of the file presently in its buffer. Thus, share access is useful for allowing several users to read different parts of the same file. It can, however, present problems when several users try to write to the file. For instance, if two users are updating a file concurrently, one could easily overwrite the other's changes when the buffer content from the first user's output is overwritten on the file by the buffer content from the second user's output.

To use Write access most effectively with shared files, specify the multiaccess option as discussed below.

To request share access for a file, use the SHR parameter in the FILE command, as follows:

```
FILE RDFILE;SHR     <----  Requests shared access
RUN RDPROG
```

Multiaccess

This option extends the features of the share access option to allow a deeper level of multiple access. Multiaccess not only makes the file available simultaneously to other accessors (in the same job or session), but permits them to use the same data pointers, blocking factor, and other file-control information. Thus, transfers to and from the file occur in the order they are requested, regardless of which program in your job or session does the requesting. When several concurrently running programs (processes) are writing to the file, the effect on the file is the same as if one program were performing all output; truly sequential access by several concurrently running programs.

NOTE Multiaccess allows the file to be shared (in all access modes) among several HPFOPEN/FOPEN requests from the same program, or from different concurrently running programs in the same job or session. Unlike share, access, however, multiaccess does not permit the file to be shared among different sessions and jobs.

Global multiaccess

This option extends the features of the multiaccess option to permit simultaneous access of a file by processes in different jobs or sessions. As in multiaccess, accessors use the same data pointer, blocking factor, and other file-control information. You can request this option as follows:

```
FILE GFILE;GMULTI     <---- Requests global multi access
RUN GPROG
```

NOTE To prohibit the use of MULTI or GMULTI access, use the NOMULTI keyword in a FILE command. When the NOMULTI keyword is used, different processes may share the data in a file, but they maintain separate buffers and pointers.

Note that it is the first accessor to a file that sets the allowable access to a file. For example, if the first accessor specifies share access, that is, the access that will be allowed to all future accessors. However, if a subsequent accessor specifies an access option that is more restrictive than the first opener's access option, it remains in effect until the user that requested it closes the file.

Sharing the File Using FLOCK and FUNLOCK

Sharing a file among two or more processes may be hazardous. When a file is being shared among two or more processes and is being written to by one or more of them, care must be taken to ensure that the processes are properly interlocked. For example, if process A is trying to read a particular record of the file, and at that time process B should execute and try to write that record, the results are not predictable. process A may see the old record or the new record, and not know whether it has read good data. If buffering is being done, please bear in mind that an output request (FWRITE) does not cause physical I/O to occur until a block is filled, which typically contains several records. A process trying to read such a file could, for example, read past the last record of the file which has been written on the disk because the end-of-file pointer is not kept in the file, but is kept in core where it can be updated quickly as writes occur. The necessary interlocking is provided by the intrinsics FLOCK and FUNLOCK, which use a resource identification number (RIN) as a flag to interlock multiple accessors.

In the simple case of a file shared between a writer process and a reader process, where the writer is merely adding records to the file, the writer calls FLOCK prior to writing each record and FUNLOCK after writing. The reader calls FLOCK prior to reading each record, and FUNLOCK after reading. If the writing process should execute while the reader is in the middle of a read, the writer will be impeded on its FLOCK call until the reader signals that it is done by calling FUNLOCK. Similarly, if the reader should execute while the writer is performing a write, the reader will be impeded on its FLOCK call until the writer calls FUNLOCK. FUNLOCK ensures that all buffers are posted on the disk so that the reading processes can see all of the data.

More complicated cases arise when a file has two or more writing processes, or when the writer may write more than one record at a time. If, for example, it should be necessary to write pairs of records, with read prohibited until both records of the pair are written, the writing process can call FLOCK before writing the first record of the pair, and FUNLOCK after writing the second.

The shared file management scheme that MPE/iX provides you through the use of the FLOCK and FUNLOCK intrinsics guarantees you exclusive access to a file being shared by a set of processes that may be located in different jobs or sessions. MPE/iX also provides you with RINs you can use to manage anything you may consider a resource to your program, be it portions of a file, a device, or a segment of code in your program. Managing shared resources with RINs is described in *Resource Management Programmer's Guide*.

For more information about the FLOCK and FUNLOCK intrinsics, consult the *MPE/iX Intrinsics Reference Manual*.

13 Maintaining File Security

MPE/iX provides two methods of establishing and maintaining file security.

- access control definitions (ACD) for file and devices
- traditional file security for disk files only

ACDs are implemented to provide a security mechanism that meets standards set forth by the National Computer Security Center. Traditional file security works through the mechanism long available on MPE systems. ACDs override any security measures implemented by traditional means. In addition, MPE/iX now provides logging facilities to track ACD security-related transactions.

ACDs are discussed first in this chapter, followed by topics relating to the traditional mechanisms of file security.

Access Control Definition Security (ACD)

MPE/iX implements a discretionary access control (DAC) mechanism that is consistent with the guidelines laid down by the National Computer Security Center.

The MPE/iX implementation, access control definitions (ACD), is a subset of the DAC mechanism. ACDs maintain a list of users and the access modes that each user has to files and devices.

ACD scope

An ACD that is associated with a file overrides the classic MPE file access matrix and lockwords, which are described later in Chapter 13 , “Maintaining File Security.”

By associating an ACD with a file or a device, the owner of the file or device may define which users have access to that file or device and which modes of access are available to other users. When a file is associated with an ACD, the ACD is put into its file label extension. The ACD contains a list of access *modes* paired with *users*.

Owners

Only those who own a file or a device may associate it with an ACD.

Files

the owner of a file is any one of these three users:

- The creator of the file with which an ACD is associated
- The user who as am capability in the account in which the file resides
- The user who has sm capability on the system in which the file resides

Devices

The System Manager (SM) is the owner of all of the devices on a system.

How acds work

When a user attempts to access a file or to acquire a device, `HPFOPEN` or `FOPEN` is called, and the system makes the following checks:

- Is the user an owner of the file or device; that is, is the user the creator of the file, the account manager (AM capability), where the file resides or the system manager (SM capability)? If *so*, permission is granted, and the checking ends.
- If *not*, is there an ACD associated with the file or device?
  - — If there is *no* ACD, the system looks for authorization in the traditional MPE/iX file access matrix and lockwords.
  - — If there *is* an ACD, the system searches, in this order, for the user:
    1. specific names (*`username.accountname`*)
    2. account groupings (*`@.accountname`*)
    3. system groupings (*`@.@`*)

       If a match *is* found, the user can access the device or file--as authorized (read, write, execute, and so on)--and no further checking is done.

       If there is *no* match, the user is denied entry, and no further checking is done.

It is important to note that if an ACD exists, the MPE/iX file access matrix and lockwords are never consulted.

ACD modes

Any device or a file can be paired with an ACD.

An ACD is associated with a file or a device by pairing access modes with users. A user is any *username.accountname* specification.

The modes of access are:

R	Read
W	Write
A	Append
L	Lock

X	Execute
NONE	None
RACD	Read and copy the ACD permission file

You could define an ACD as follows:

```
ACD = (R,W:MGR.ACCTING, DENNIS.LEE; R:@.PAYROLL; A:@.@)
```

The users `MGR.ACCTING` and `DENNIS.LEE` can read and write to the file associated with this example ACD. Anyone in the `PAYROLL` account can read it, and anyone on the system can append to it. For example, no one but `DENNIS.LEE` and the owners can overwrite the file, and only the owners can lock it.

NOTE If an ACD exists and if you are not explicitly given permission to access a file or a device, you do not have access.

Managing ACDs with commands and intrinsics

Use MPE/iX commands to manage ACDs interactively, through the command interpreter. Use MPE/iX intrinsics to manage ACDs in a program.

Commands

These MPE/iX commands accept ACD-related parameters or incorporate ACD associations in their operation:

`ALTSECT`	Permits the addition, creation, deletion, modification, copying, and listing of ACD attributes.
`COPY`	Always copies the ACD associated with the source file to the target file, if an ACD is present.
`FCOPY`	Permits copying ACD attributes.
`FILE`	Permits the equation of one file/device-ACD specification to another file/device-ACD specification.
`LISTFILE`	Permits the listing of the ACD attributes associated with a file or device.
`RELEASE`	Returns a warning when an ACD is associated with a file.
`RESTORE`	Accomodates ACDs.
`SECURE`	Returns a warning when an ACD is associated with a file.
`SHOWDEV`	Permits the listing of ACD attributes associated with a device.
`STORE`	Accomodates ACDs.

Detailed discussions of these commands are found in the *MPE/iX Commands Reference Manual*.

Intrinsics

`HPACDPUT`	Permits the addition, creation, deletion, modification, copying, and listing

of ACD attributes.

HPACDINFO Returns security attributes.

HPFOPEN Permits the creation of of an ACD.

The intrinsic FOPEN cannot be modified to give it the option of creating an ACD. You must use HPFOPEN.

Detailed discussions of these intrinsics are found in the *MPE/iX Intrinsics Reference Manual*.

Preserving ACDs

Device ACDs are *not* permanent objects; you must redefine them every time that the system is rebooted. The easiest way to do this is to put ALTSEC commands into the SYSSTART file, either directly or in a command file.

File ACDs *are* permanent objects; they do survive a reboot. When you store files to tape, FCOPY and STORE save the files' ACDs, too–unless you specify otherwise. If you are not an owner of the file and you do not have RACD permission, you get an error if you try to copy the ACD. Instead, choose the NOACD parameter.

Managing ACDs

You may manage ACDs interactively through MPE/iX commands or programmatically through MPE/iX intrinsics.

Creating ACDs

Command	Intrinsic	Purpose
ALTSEC		Create an ACD for an existing device or file
	HPACDPUT	Create an ACD for an existing device or file

Examples

To assign Read access to user SAM.DOE, Write access to JOE.DOE, no access (None) to all users in the DESIGN account, and Execute access to all users in all accounts (except those users in the DESIGN account, enter this:

```
ALTSEC
FILEA.XX.DESIGN;NEWACD=(R:SAM.DOE;W:JOE.DOE;NONE:@.DESIGN;X:@.@)
```

To add an ACD that prevents any user except OPERATOR.SYS from accessing LDEV 7 (a tape drive), enter this:

```
ALTSEC 7,LDEV;NEWACD=(R,W:OPERATOR.SYS)
```

The user must have SM capability to do this.

This short program uses HPACDPUT in creating an ACD for a file called TARGET:

```
program acdput(input, output);
```

```
var
  status     : integer;
  filename   : packed array [1..28] of char;
  ACD        : packed array [1..256] of char;

procedure HPACDPUT;intrinsic;

begin
filename := 'TARGET';
ACD      := '(x:@.@;r,w:mgr.sys)'
ACD[20] := #m;
HPACDPUT(status, 1, filename, 20, ACD)
if status <> 0 then
   writeln('HPACDPUT failed.  Status = ', status);
end.
```

When you create a new file with the COPY, FCOPY, STORE, or RESTORE commands, you can use the command parameters to create the ACDs for the new file.

- COPY

  The COPY command automatically copies any ACD attributes from the source file to the target file, provided that the user is an owner of the source file or has RACD access to that file.

  ```
  COPY FILEA,FILEB
  ```

- FCOPY

  The ;COPYACD parameter of the FCOPY command permits the user to copy a file and its ACD, provided that the user is an owner or has RACD permission.

  ```
  FCOPY <;fcopycommand>;COPYACD
  ```

- STORE

  To store all of the files on a system to tape, including their ACDs, enter this:

  ```
  FILE T;DEV=TAPE
  STORE @.@.@;*T;COPYACD
  ```

  COPYACD is the default. You must have access to any ACD-protected files being stored.

  — SM and OP can store any ACD-protected file on the system.

  — AM can store any ACD-protected file in the manager's account.

  — Users can store any ACD-protected files that they own, provided that they have Read access to the file and RACD access to the file if ;COPYACD is specified.

  — Others can store ACD-protected files for which they have RACD permission, provided that they have Read access to the file and RACD access to the file if ;COPYACD is specified.

  — You must have PM access to a PM file in order to store it.

- RESTORE

To restore all of the files on tape and copies the ACD attributes of the file to disk, enter this:

```
FILE T;DEV=TAPE
RESTORE *T;@;KEEP;SHOW;COPYACD
```

COPYACD is the default. NOACD prevents the copying of the ACD attributes.

— SM and OP can restore any ACD-protected file on the system.

— AM can restore any ACD-protected file in the manager's account.

— Users can restore any ACD-protected files that they own, provided that they have Read access to the file and RACD access to the file if ;COPYACD is specified.

— Others can restore ACD-protected file for which they have RACD permission, provided that they have Read access to the file and RACD access to the file if ;COPYACD is specified.

— You must have PM access to a PM file in order to store it.

Listing ACDs

Command	Intrinsic	Purpose
LISTFILE		Show ACDs for files
SHOWDEV		Show ACDs for devices
	HPACDINFO	Show ACDs for files and devices
	HPACDPUT	Show ACDs for files and devices

Examples

The LISTFILE command with option 4 shows the ACD status of a file in this fashion:

```
LISTFILE FILEA,4
****************************
FILEA.XX.DESIGN

SYSTEM     READ:      ANY
SECURITY--WRITE:      AC
 (ACCT)    APPEND:    AC
           LOCK:      AC
           EXECUTE:   ANY

SYSTEM     READ:      GU
SECURITY--WRITE:      GU
 (GROUP)   APPEND:    GU
           LOCK:      GU
           EXECUTE:   GU

SYSTEM     READ:      ANY           FCODE:  0
SECURITY--WRITE:      ANY           CREATOR:  **
 (FILE)    APPEND:    ANY           LOCKWORD:  **
           LOCK:      ANY         **SECURITY IS ON
           EXECUTE:   ANY         **ACD EXISTS
```

```
FOR XX.DESIGN: NONE
```

(Other ACD status reports are `NO ACD` and `ACD CORRUPTED`.)

The `LISTFILE` command with option `-2` gives a detailed ACD report on a file in this fashion:

```
LISTFILE FILEA,-2

FILE = FILEA      ************** ACD ENTRIES **************

                  SAM.DOE             : R
                  JOE.DOE             : W
                  @.DESIGN            : NONE
                  @.@                 : X
```

The `SHOWDEV` command displays the ACD attributes of a device in this fashion:

```
SHOWDEV 14;ACD
LDEV  AVAIL      OWNERSHIP      VOLID       DEN   ASSOCIATION

 14   SPOOLED      SPOOLER OUT
      ACD ENTRIES: @.@ : R,W,X
```

This short program uses `HPACDINFO` to retrieve the number of entries and first user in the ACD of a file called `TARGET`:

```
program acdinfo(input, output);

type
  shortint = -32768..32767;

var
  status      : integer;
  filename    : packed array [1..28] of char;
  numentry    : shortint;
  firstuser   : packed array [1..18] of char;

procedure HPACDINFO;intrinsic;

begin
filename := 'TARGET';
HPACDINFO(status, 1, filename, 21, numentry, firstuser);
if status = 0 then
  begin
  writeln('Number of Entires:  ', numentry:1);
  writeln('First UserSpec    :  ', firstuser);
  end;
end.
```

Copying ACDs

Command	Intrinsic	Purpose
;COPYACD parameter of the ALTSEC command		Copy an ACD from one file to another

Examples

To copy the ACD associated with FILEB to FILEA, enter this:

```
ALTSEC FILEA.XX.DESIGN;COPYACD=FILEB.XX.DESIGN
```

Only an owner, or a user granted RACD (read ACD) authorization, can copy the ACD from FILEB.

To copy the ACD attributes of LDEV 7 to LDEV 23, enter this:

```
ALTSEC 23,LDEV;COPYACD=7,LDEV
```

Only users with SM capability may do this. By definition, users having SM capability are owners of all the files and devices on a system. Those users may give themselves access to any file or device on the system.

Modifying ACDs

Command	Intrinsic	Purpose
ALTSEC		To change an ACD
	HPACDPUT	To change an ACD

Adding ACD pairs

To confer Read access on JOE.DESIGN for FILEA, enter this:

```
ALTSECT FILEA.XX.DESIGN;ADDPAIR=(R:JOE.DESIGN)
```

Replacing ACDs

To change the (previous) Read access for SAM.DOE to Write access enter this:

```
ALTSECT FILEA.XX.DESIGN;REPPAIR(W:SAM.DOE)
```

To assign Read and Write access to SAM.DOE, do this:

```
ALTSEC FILEA.XX.DESIGN;REPPAIR(W,R:SAM.DOE)
```

Deleting ACDs

Command	Intrinsic	Purpose
ALTSEC		To delete an ACD
	HPACDPUT	To delete an ACD

To remove @.DESIGN from the ACD attributes of FILEA, enter this:

```
ALTSEC FILEA.XX.DESIGN;DELPAIR(NONE:@.DESIGN)
```

To deny OPERATOR.SYS any access to LDEV 7, enter this:

```
ALTSEC 7,LDEV;DELPAIR=(R,W:OPERATOR.SYS)
```

Only an owner can delete an ACD associated with a file. Only the system manager can delete an ACD associated with a device..

Migrating ACDs

Device ACDs should not be migrated, because they are tied to their system's configuration.

You can move file ACDs between MPE V/E and MPE/iX by using the STORE and RESTORE commands, where COPYACD is the default.

These are the steps CM RESTORE takes during forward migration:

1. reads the MPE V/E store format.
2. calls a routine to convert it to MPE/iX internal format.
3. calls the file label extension write routine, which puts the ACD into effect.

These are the steps CM STORE takes during backward migration:

1. reads the ACD from the security file label extension.
2. calls a routine to convert it into MPE V/E format.
3. writes it out to the STORE tape.

Be aware that MPE/iX allows more *user-mode* pairs than MPE V/E does.

You must have authorization to use the ;COPYACD parameter of the STORE and RESTORE commands. If you are not an owner of the file or do not have RACD permission, you get an error. The STORE command checks the ACD on disk for permission. RESTORE checks the ACD from the tape.

For more details, refer to the *MPE/iX Commands Reference Manual* and the *MPE/iX Intrinsics Reference Manual*.

Logging system events

The following list shows the types of logs that you can request.

Table 13-1. SYSGEN System Logging

System Log Events	Event Type
System logging enabled	100
System up record	101
Job initiation record	102
Job termination record	103
Process termination record	104

Table 13-1. SYSGEN System Logging

System Log Events	Event Type
File close record	105
System shutdown record	106
Power failure record	107
Spooling log record	108
I/O error record	111
Physical mount/dismount	112
Logical mount/dismount	113
Tape labels record	114
Console log record	115
Program file event	116
New commercial spooling	120
Architected interface	130
Password changes	134
System logging configuration	135
Restore logging	136
Printer access failure	137
ACD changes	138
Stream initiation logging	139
User logging	140
Process creation	141
Chgroup record	143
File open record	144
Maintenance request log	146
Diagnostic information record	150
High priority machine check	152
Low priority machine check	152
CM file close record	160

All log information is kept in records. Each record begins with a standard header and ends with identification information. The information between is different for each log type. The LOGTOOL utility has a standard format to display information.

Log of system logging configuration

This log gives you an audit trail of changes to the logging configuration. This log is initially enabled (ON). The following is the log record format:

Table 13-2. Type 135 Record Format

Length, in 16-bit words	Record Content
1	Record type (135)
1	Record length
1	Process identification number
3	Time stamp
2	Job type/job number
1	(Reserved)
1	LDEV number
4	System logging masking words
8	User name
8	Group name
8	Account name
8	job or session name

Log of restore

This log traces file restorations. Files can be restored from tape or serial disk to the system. This log type is initially disabled (OFF). It can be enabled by SYSGEN followed by a `START` command. The following is the log record format:

Table 13-3. Type 136 Record Format

Length, in 16-bit words	Record Content
1	Record type (136)
1	Record length
1	Process identification number
3	Time stamp
2	Job type/job number
8	File name
8	File group
8	File account

Table 13-3. Type 136 Record Format

Length, in 16-bit words	Record Content
8	Creator
17	Volume identification
1	Access type
8	User name
8	Group name
8	Account name
8	job or session name

Log of printer access failure

This log keeps track of failed attempts attaching spool files to printers. New spool files, which are logged by FOPEN as event #144, are not logged here.

This log is initially disabled, but can be enabled by SYSGEN followed by a START command.

Table 13-4. Type 137 Record

Length, in 16-bit words	Record Content
1	record type (137)
1	record length
1	process identification number
3	time stamp
2	job type/job number
2	creator job number
8	creator job name
8	creator user name
8	creator account name
25	spool file name
8	target device name/class
1	(reserved)
2	file size
1	status
8	user name

Table 13-4. Type 137 Record

Length, in 16-bit words	Record Content
8	group name
8	account name
8	job or session name

Log of stream initiation

This log records the name of a streamed job, its number, the user that initiates it (and the logon), and the scheduled date and time.

This log is initially disabled, but can be enabled by SYSGEN followed by a `START` command.

Table 13-5. Type 139 Record

Length, in 16-bit words	Record Content
1	Record type (139)
1	record length
1	process identification number
3	time stamp
2	job type/job number
1	input LDEV
25	job file name
2	job logon job or session number
8	job logon user
8	job logon group
8	job logon account
8	job name
2	input spool file id
1	scheduled date
2	scheduled time
8	user name
8	group name
8	account name
8	job or session name

Log of user logging

This log keeps a record of all OPENLOG and CLOSELOG intrinsic calls. The system manager can use it to see who accesses, or tries to access, the user logging facility.

This log is initially disabled, but can be enabled by SYSGEN followed by a START command.

Table 13-6. Type 140 Record Format

Length, in 16-bit words	Record Content
1	record type (140)
1	record length
1	process identification number
3	time stamp
2	job type/job number
25	program file name
4	intrinsic
2	index
4	log id
1	mode
1	status
8	user name
8	group name
8	account name
8	job or session name

The LOG ID field in the log record is "XXXXXX" for CLOSELOG intrinsic when the index is bad.

Log of process creation

You can use this log to record all process creations. This log is initially disabled, but can be enabled by SYSGEN followed by a START command.

Table 13-7. Type 141 Record

length, in 16-bit words	Record Content
1	record type (141)
1	record length
1	process identification number

Table 13-7. Type 141 Record

length, in 16-bit words	Record Content
3	time stamp
2	job type/job number
25	file name
1	(reserved)
2	priority
2	process space id
4	parent PID
2	NM_Heap_Size
2	capabilities mask*
8	(reserved)
8	user name
8	group name
8	account name
8	job or session name

*The capabilities mask is read as follows:

```
      User               File access          Program/group
bit     capability    bit    capability      bit    capability
 0         SM          6        CV            23       BA
 1         AM          7        UV            24       IA
 2         AL          8        LG            25       PM
 3         GL          9        SP            28       MR
 4         DI         10        PS            30       DS
 5         OP         11        NA            31       PH
                      12        NM
                      13        CS
                      14        ND
                    15        SF
```

Logging a specific user

The LOGTOOL utility command `LIST` shows you the output of log records in a standard format. If you like, you can filter the output of LOGTOOL utility to show you information about only a specific user or users. The syntax for this is shown below.

```
LIST {LOG=log_name}[;JSNAME=job or session_name
                    ;USER=user_name
                    ;ACCOUNT=account_name  ][...]
```

The input for these commands should be no longer than 80 characters. Default for all

parameters is the wildcard @.

For example, to select log records from log files 1 through 5, with log information about password changes (log type 134), and user identification JTEST,MARIA.PAYROLL, you would enter the following.

```
>LIST LOG=1/5;TYPE=134;JSNAME=JTEST;USER=MARIA;ACCOUNT=PAYROLL
```

This selection option is valid for the log types listed below:

102, job initiation

103, job termination

104, process termination

105, file close (also 160)

108, spooling log

112, physical mount/dismount

113, logical mount/dismount

114, tape labels

115, console log

116, program file event

120, new commercial spooling

130, architecture interface

134, password change

135, system logging configuration

136, restore

137, printer access failure

138, ACD changes

139, stream initiation

140, user logging access

141, process initiation

143, change group

144, file open

Logging file security related events

MPE/iX permits logging of system and user events. The events that relate directly to file security are:

- password changes (event type 134)
- printer access failure (event type 137)
- ACD changes (event type 138)

Logging begins whenever the system is rebooted; however, not all events are automatically enabled. Some, including those listed above, are initially disabled. You can, however, request that a new file be started.

To keep a certain type of log, the system operator or system manager must change its status to ON (configure it) in SYSDIAG. To see log records displayed, call the LOGTOOL utility from SYSGEN.

For a discussion of these and other logging facilities, consult these topics in *Performing System Operator Tasks*: SYSDIAG, the LOGTOOL utility, and SYSGEN System Logging.

Log of password changes

System logging records when a user, group, or account password is changed by an MPE/iX command or a utility program. This log is initially disabled (OFF).

The information recorded in this logging includes

- header
  - — record type
  - — record length
  - — time stamp
  - — job or session number
  - — PIN
- Log information
  - — the identification of the user who changed a password: job or session name, user name, group name, and account name
  - — the identification of a user whose password was changed: user name, group name, and account name whenever the affected password changes
  - — input logical device number from which the password was changed
  - — program file name from which password change was executed
  - — type changed: 1 = user, 2 = group, 4 = account

In this example, JOHN.PAYROLL,DOE, job or session name JREPORT, successfully changed the account password for PAYROLL through the command excutor. The change was made from LDEV 21.

The LOGTOOL utility formats the following layout after the standard header:

```
TARGET USER:                        TARGET GROUP:
TARGET ACCOUNT:     PAYROLL         TYPE CHANGED:     ACCOUNT
LDEV:               21
EXECUTED FROM:      CI.PUB.SYS
USER:               JOHN            GROUP:            DOE
ACCOUNT:            PAYROLL         JSNAME:           JREPORT
```

The following is the log record format:

Table 13-8. Type 134 Record Format

length, in 16-bit words	Record Content
1	record type (134)
1	record length
1	process identification number
3	time stamp
2	job type/job number
8	target user name
8	target group name
8	target account name
1	type changed
1	input LDEV number
25	executed from
3	(reserved)
8	user name
8	group name
8	account name
8	job or session name

NOTE The PASSWORD command, allows all users to change their own passwords. In the past, only system managers and account managers could change any passwords.

Log of ACD changes

This log type is activated when ACDs are changed (created, deleted, copied, or modified) with MPE/iX commands or intrinsics. The log is initially disabled (OFF).

The information recorded in this logging includes

- header
  - record type
  - record length
  - time stamp
  - job or session number

  — PIN

- log information

  — the identification of the user who changed the ACD: job or session name, user name, group name, and account name

  — the object type and object name whose ACD was changed

  — the object type and object name from which the ACD was copied

  — the type of change to the ACD: create, add pair, replace pair, copy, delete pair, delete

  — the program file name from which the ACD change was executed.

  — status returned (HPE status)

In this example, user JOHN.PAYROLL,DOE, with job or session name JREPORT, successfully created an ACD for a file called FTEST.TESTGP.PAYROLL, using the command executor.

The LOGTOOL formats the following layout after the standard header:

```
TARGET OBJECT:          FTEST.TESTGP.PAYROLL
SOURCE OBJECT:
FUNCTION:               CREATE
EXECUTED FROM:          CI.PUB.SYS
STATUS                  SUCCESSFUL
USER                    JOHN              GROUP:         DOE
ACCOUNT:                PAYROLL           JSNAME:        JREPORT
```

The following is the log record format:

Table 13-9. Type 138 Record Format

length, in 16-bit words	Record Content
1	Record type (138)
1	Record length
1	Process identification number
3	Time stamp
2	Job type/job number
25	Target object name
25	Source object name
4	Function
25	Executed from
2	Status
8	User name
8	Group name

Table 13-9. Type 138 Record Format

length, in 16-bit words	Record Content
8	Account name
8	job or session name

Traditional Mechanism for File Security

The traditional security mechanism (file access matrix and lockwords) associates with each account, group, and individual files a set of security provisions that specifies any restrictions on access to the files in that account or group, or to that particular file.

NOTE These provisions apply to disk files only. If a file is protected by the traditional security mechanism *and* by an ACD definition, the ACD definition overrides the traditional security mechanism. ACD security mechanism are discussed at the beginning of this chapter under <Undefined Cross-Reference>.

These restrictions are based on two factors:

- modes of access--reading, writing, or saving, for example.
- types of user--users with account librarian (AL) or group librarian (GL) capability, or creating users, for example, to whom the access modes specified are permitted

The security provisions for any file describe what modes of access are permitted to which users of that file.

Specifying and restricting file access by access mode

When a program opens or creates a file, it can define the way that the file can be accessed by specifying a particular access mode (such as Read-only, Write-only, Update, and so forth) for the file. These specifications apply to files on any device and can be changed or overridden only by yourself, as the creator of the file. They are discussed in the following paragraphs. In addition, for files on disk, a program can also restrict access so that only one access attempt (`HPFOPEN/FOPEN` call) or process (running program) can open it at one time, or can allow it to be shared among several accessors.

The access types that can be specified by a program are listed in Table 13-10. on page 180

When specifying the access mode for a file, it is important to realize where the current end-of-file is before and after the file is opened, and where the logical record pointer indicates that the next operation will begin. These factors depend upon the access mode that you select. Because they are best explained by example, the effects of each access mode upon these factors are summarized in Table 13-1. on page 167 for a sample file. This

file contains 10 logical records of data (numbered 0 through 9). The table shows that the current end-of-file (EOF) lies at Record 10 before the file is opened, indicating that if another record were appended to the file, that would be the eleventh record. When you open the file in the Write-only mode, however, all records presently in the file are deleted and the logical record pointer and current EOF move to record 0. Now when you write a record to the file, this will be the first record in that file.

Suppose that you are running a program that opens a magnetic tape file for Write-only access, but you wish to append records to that file rather than to delete existing records.

You can override the programmatic specifications by using the `FILE` command to request Append access to the file, as follows:

```
FILE TASK; DEV=TAPE; ACC=APPEND
RUN PROGN                        \
                                  Requests append access
```

Table 13-10. Traditional File Access Mode Types

ACCESS MODE	:FILE\ PARAMETER	DESCRIPTION
Read-only	`IN`	Permits file to be read but not written on. Used for device files, such as card reader and paper tape reader files, as well as magnetic tape, disk, and terminal output files.
Write-only	`OUT`	Permits file to be written on but not read. Any data already in the file is deleted when the file is opened. Used for device files, such as card punch and line printer, as well as tape, disk, and terminal output files.
Write-SAVE	`OUTKEEP`	Permits file to be written on but not read, allowing you to add new records both before and after current end-of-file indicator. Data is not deleted, but a normal write replaces it.
Append-only	`APPEND`	Permits information to be appended to file, but allows neither overwriting of current information nor reading of file. Allows you to add new records after current end-of-file indicator only. Used when present contents of file must be preserved.
Read/Write	`INOUT`	Permits unrestricted input and output access of file; information already on file is saved when the file is opened. (In general, combines features of `IN` and `OUTKEEP`.)
Update	`UPDATE`	Permits the use of `FUPDATE` intrinsic to alter records in file. Record is read into your data stack, altered, and rewritten to file. All data already in file is saved when the file is opened.

Suppose that you run a program that opens a disk file for write-only access, copies records into it, and closes it as a permanent file. Under the standard file system security provisions, the access mode is automatically altered so that the file permits the read, write, and append access modes (among others). Now, suppose that you run the program a second

time, but wish to correct some of the data in the file rather than delete it. You could use the `FILE` command to override the programmatic specification, opening the file for update access:

```
FILE REPFILE; ADD=UPDATE
RUN PROGN                    \
                              Requests update access
```

Table 13-11. Effects of Access Modes

ACCESS MODE	CURRENT EOF	LOGICAL RECORD POINTER	EOF AFTER OPEN
Read-only	10	0	10
Write-only	10	0	0
Write-SAVE	10	0	10
Append	10	10	10
Read/Write	10	0	10
Update	10	0	10

Consider a program that reads input from a terminal (file name `INDEV`) directs output to a line printer (`OUTDEV`). You can redirect the output so that it is transmitted to the terminal by entering:

```
FILE INDEV; DEV=TERM; ACC=INOUT    <---- Respecifies INDEV for
                                          both input and output access

FILE OUTDEV=*INDEV    <---- Equates INDEV to OUTDEV

RUN PROGO    <---- Runs program
```

Specifying and restricting file access by type or user

Restrictions on who can access a file are established when the file is created according to the default prescribed for the group and account where the file resides. The capabilities of the user who accesses a file may determine the security restrictions that apply to him. The types of users recognized by the MPE/iX security system, the mnemonic codes used to reference them, and their complete definitions are listed in Table 13-12. on page 181

Table 13-12. User Type Definitions (Traditional Security)

USER TYPE	MNEMONIC CODE	MEANING
Any User	ANY	Any user defined in the system; this includes all categories defined below.
Account Librarian User	AL	User with Account Librarian capability, who can manage certain files within his account that may or may not all belong to one group.

Table 13-12. User Type Definitions (Traditional Security)

USER TYPE	MNEMONIC CODE	MEANING
Group Librarian User	GL	User with Group Librarian capability, who can manage certain files within his home group.
Creating User	CR	The user who created this file.
Group User	GU	Any user allowed to access this group as his logon or home group, including all GL users applicable to this group.
Account Member	AC	Any user authorized access to the system under this account; this includes all AL, GU, GL, and CR users under this account.

Users with system manager or account manager capability bypass the standard security mechanism. A system manager has unlimited file access to any file in the system (R,A,W,L,X:ANY), but can save files only in his own account (S:AC); an account manager user has unlimited access to any file within the account (R,A,W,L,X,S:ANY). One exception is that in order to access a file with a negative file code (a privileged file), the account manager must also have the privileged mode (PM) capability.

The user-type categories that a user satisfies depend on the file he is trying to access. For example, a user accessing a file that is not in his home group is not considered a group librarian for this access even if he has the group librarian user attribute.

NOTE In addition to the above restrictions in force at the account, group, and file level, a file lockword can be specified for each file. Users then must specify the lockword as part of the file name to access the file.

The security provisions for the account and group levels are managed only by users with the system manager and the account manager capabilities respectively, and can only be changed by those individuals.

Account-level security

The security provisions that broadly apply to all files within an account are set by a system manager user when creating the account. The initial provisions can be changed at any time, but only by that user.

At the account level, five access modes are recognized:

- reading (R)
- appending (A)
- writing (W)
- locking (L)
- executing (X)

Also at the account level, two user types are recognized:

- any user (ANY)
- account member (AC)

If no security provisions are explicitly specified for the account, the following provisions are assigned by default:

- For the system account (named `SYS`), through which the system manager user initially accesses the system, reading and executing access are permitted to all users; appending, writing, and locking access are limited to account members.

NOTE Symbolically, these provisions are expressed as follows:

(R,X:ANY;A,W,L:AC)

In this format, colons are interpreted to mean, "...is permitted only to..." or "... is limited to ...." Commas are used to separate access modes or user types from each other. Semicolons are used to separate entire access mode/user type groups from each other.

- For all other accounts, the reading, appending, writing, locking, and executing access modes are limited to account members (R, A, W, L, X: AC).

Group-level security

The security provisions that apply to all files within a group are initially set by an account manager user when creating the group. they can be equal to or more restrictive than the provisions specified at the account level. (The group's security provisions also can be less restrictive than those of the account–but this effectively results in equating the group restrictions with the account restrictions, since a user failing security checking at the account level is denied access at that point and is not checked at the group level.) The initial group provisions can be changed at any time, but only by an account-managing user for that group's account.

At the group level, six access modes are recognized:

- reading (R)
- appending (A)
- writing (W)
- locking (L)
- executing (X)
- saving (S)

Also at the group level, five user types are recognized:

- any user (ANY)
- account librarian user (AL)
- group librarian user (GL)
- group user (GU)

- account member (AC)

If no security provisions are explicitly specified, the following provisions apply by default:

- For a public group (named PUB), whose files are normally accessible in some way to all users within the account, reading and executing access are permitted to all users; appending, writing, saving, and locking access are limited to account librarian users and group users (including group librarian users). (R, X: ANY; A, W, L, S: AL, GU).
- For all other groups in the account, reading, appending, writing, saving, locking, and executing access are limited to group users. (R, A, W, L, X, S: GU).

File-level security

When a file is created, the security provisions that apply to it are the default provisions assigned by MPE/iX at the file level, coupled with the user-specified or default provisions assigned to the account and group to which the file belongs. At any time, however, the creator of the file (and only this individual) can change the file-level security provisions, as described in the following pages; thus, the total security provisions for any file depend upon specifications made at all three levels, the account, group, and file levels. A user must pass tests at all three levels–account, group, and file security, in that order–to successfully access a file in the requested mode.

If no security provisions are explicitly specified by the user, the following provisions are assigned at the file level by default:

- For all files, reading, appending, writing, locking, and executing access are permitted to all users. (R, A, W, L, X: ANY).

Because the total security for a file always depends on security at all three levels, a file not explicitly protected from a certain access mode at the file level may benefit from the default protection at the group level. For example, the default provisions at the file level allow the file to be read by any user–but the default provisions at the group level allow access only to group users; thus, the file can be read only by a group user.

In summary, the default security provisions at the account, group, and file levels combine to result in overall default security provisions as listed in Table 13-13. on page 185 Stated another way, when the default security provisions are in force at all levels, the standard user (without any other user attributes) has:

- unlimited access (in all modes) to all files in his logon group and home group
- reading and executing access (only) to all files in the public group of his account and the public group of the system account

The important file security rules may be defined as follows:

- Users can create files in their own accounts.
- Only the creator can modify a file's security.
- If a lockword is present on a file, then it is required in order to access the file.
- Account managers have unlimited access to the files within their accounts.
- System managers have unlimited access to any file, but can save files only in their

account.

Table 13-13. Default Security Provisions (Traditional)

FILEREFERENCE	FILE	ACCESS\PERMITTED	SAVE ACCESS\TO GROUP
filename.PUB.SYS	Any file in public group of system account	(R,X:ANY; W:AL,GU)	AL,GU
filename.*groupname*.SYS	Any file in any group in system account	(R,W,X:GU)	GU
filename.PUB.*accountname*	Any file in public group of any account	(R,X:AC; W:AL,GU)	AL,GU
filename.*groupname*.*accountname*	Any file in any group in any account	(R,W,X:GU)	GU

Changing security provisions of disk files

The security provisions for both the account and group levels are managed only by users with the system manager capability, while group level security is managed by users with account manager capability. Even if you have only standard capabilities (IA, BA, SF), you can change the security provisions for any disk file that you have created. You do this by using the ALTSEC command, which permanently deletes all previous provisions specified for this file at the file level, and replaces them with those defined as the command parameters. This command does not, however, affect any account-level or group-level provisions that may cover the file. Furthermore, it does not affect the security provided by the lockword (if one exists).

For example, suppose that you want to alter the security provisions for the file FILEX to permit the ability to read, execute, and append information to the file only to the creating user and the logon or home group users. You can do this with the following ALTSEC command:

```
ALTSEC FILEX; (A,R,X:CR,GU)
```

Any parameters not included in the ALTSEC command are cleared.

To restore the default security provisions to this file, you would enter:

```
ALTSEC FILEX
```

Suppose that you have created a file named FILEZ for which you have allowed yourself program-execute access only. You now wish to change this file's security provisions so that any group user can execute the program stored within it, but only the group librarian can read and write on it. Even though you do not have Read or Write access to the file, you can still alter its security provisions by entering:

```
ALTSEC FILEZ; (X:GU;R,W:GL)
```

You always retain the ability to change the security provisions of a file that you have created, even when you are not allowed to access the file in any mode; thus, you can even change the provisions to allow yourself access.

Suspending and restoring security provisions

You may temporarily suspend the suspending and restoring security:files| security restrictions on any disk file that you create. This allows the file to be accessed in any mode by any user; in other words, it offers unlimited access to the file. You suspend the security provisions by entering the RELEASE command. (File lockword protection, however, is not removed by this command.) The RELEASE command does not modify the file security settings recorded in the system; it bypasses them temporarily. The RELEASE command remains in effect until you enter the SECURE command in this or a later job or session.

To release the security provisions for the file named FILESEC in your logon group, enter:

```
RELEASE FILESEC
```

If the file has a lockword and that you wish to remove that as well as all account-level, group-level, and file-level security provisions, you must use the RENAME command, as well as the RELEASE command:

```
RENAME FILESEC/LOCKSEC,FILESEC   <---- Removes lockword
RELEASE FILESEC   <---- Removes security provisions
```

To restore the security provisions of a file, use the SECURE command. For example:

```
SECURE FILESEC
```

The original security restrictions for the file will be in effect.

14 Getting File Information

MPE/iX provides a number of commands and intrinsics that enable you to obtain information about your files. You can use the commands and intrinsics described in this chapter to obtain file information for a variety of purposes.

This chapter is divided into two main sections:

- General File Information covers the commands and intrinsics that you use to obtain information concerning the physical and operational characteristics of your file (defined by device dependencies, a disk file's label, `FILE` commands, `HPFOPEN/FOPEN` intrinsic parameters, and file system defaults), as well as access-dependent details about a currently opened file (including EOF and logical record marker locations). Commands and intrinsics described in this section are:

  `LISTFILE` command

  `LISTEQ` command

  `[CMD] INFO` command

  `FFILEINFO` intrinsic

  `FGETINFO` intrinsic

  `FLABELINFO` intrinsic

  `FRELATE` intrinsic

- Error information covers the intrinsics that you use specifically to handle file system errors–to identify an error and to display error information at the terminal. This can include a description of the error condition returned by the last file access intrinsic call. Intrinsics described in this section are:

  `FCHECK` intrinsic

  `FERRMSG` intrinsic

  `PRINTFILEINFO` intrinsic

A number of commands and intrinsics return the same information. Which one you use in a certain situation is by the context and by the purpose for which you wish to use the information.

Displaying General File Information

Some questions that you may wish to answer in this section include

- Does a file by this name exist in my account or group?
- How large is this file?
- When and by whom was this file created?
- What security provisions exist for this file?

You can use this information in an interactive context, or you can use commands and intrinsics within an executing program to obtain and utilize the information.

You use:	To obtain information about:
LISTFILE	Characteristics of a permanent file
LISTFILE...;TEMP	Characteristics of a temporary file
LISTEQ	File equations in effect for current job or session

Displaying permanent file information with LISTFILE

The LISTFILE command is one of the most widely used commands in MPE/iX. You use LISTFILE to display information about one or more permanent files that you specify. LISTFILE has parameters that allow you to:

- specify a set of permanent or temporary files that you wish information about
- specify the amount, or level, of file information that you wish to see
- specify a file where the LISTFILE output is written to

Specifying a file reference

If you do not indicate otherwise, LISTFILE displays information about all of the permanent files located in your logon group. You can optionally reference a file (or files) by specifying a file name and qualifying it with the appropriate group and/or account name. For example, if your logon group and account are MYGROUP.MYACCT, the following commands will return information about the same file:

```
LISTFILE MYFILE
LISTFILE MYFILE,MYGROUP
LISTFILE MYFILE,MYGROUP,MYACCT
```

In addition, you can use MPE/iX wildcard characters, in conjunction with a qualified file reference, to specify a set of files you want LISTFILE to display information about. The wildcard characters you can use are

@ = zero or more alphanumeric characters

= a single numeric character

? = a single alphanumeric character

Specifying the list level

If you do not indicate otherwise, `LISTFILE` displays only the unqualified file name. You can optionally direct `LISTFILE` to display more detailed information about the file(s) that you specify, depending upon the parameter value that you specify in the command, as listed in Table 14-1. on page 189

Table 14-1. Format Selection

Option	Name	Displayed Information
–2	ACD	Displays the file's ACD (access control definition). System managers can view the ACD for any file. Account managers can view the ACD for files in that account. File creators can view the ACD for their files. Other users can view an ACD only if that ACD specifies that the user has RACD (read ACD) access.
–1	LABEL	Displays the hexadecimal listing of the file label, including all lockwords. This level is available only to system managers and account managers.
0	FILES	Shows only the file name. This is the default.
1	SUMMARY	Displays the file name, file code, record size, record format, and other file characteristics such as ASCII or binary records, carriage-control option, current end-of-file location, and the maximum number of records allowed in the file.
2	DISC	Displays the file name, file code, record size, record format, and other file characteristics such as ASCII or binary records, carriage-control option, current end-of-file location, and the maximum number of records allowed in the file. It also displays the blocking factor, number of sectors in use, number of extents currently allocated, and the maximum number of extents allowed. LISTF, 2 also displays KSAM XL file types with "K", and KASAM64 with “K”..
3	DETAIL	Displays the file name, record size, extent size, number of records, access rights for the user, and other file characteristics including the date created, modified, and last accessed. The creator, lockword, and label address are omitted. These can be obtained by specifying -3 if you have AM capability (for files in your account) or SM capability (for any file on the system).
4	SECURITY	Displays the security matrix for the file. This includes account-, group- and file-level security and the access rights for the user. If an access control definition exists (ACD), a message stating that fact is displayed.
5	DATA	Shows `LISTFILE,3` data and all file-specific data in `LISTF`, 3 type format (that is, KSAM and `SPOOL`).
6	QUALIFY	Shows only fully qualified file name.
7	UNIQUE	Shows all file specific data in `LISTFILE,3` type format, but does not show `LISTFILE,3` data.

Table 14-1. Format Selection

Option	Name	Displayed Information
8	ACCESS	Shows all accessors for the file
9	LOCKS	Shows all format 8 data plus more details about the proecsses accessing the file, including locking data.
10	SUMMARY WIDE	The same basic information as in the SUMMARY option, but in a wider format to allow for larger values.
11	DISCWIDE	The same basic information as in the DISC option, but in a wider format to allow for larger values.

Specifying an alternate output file

If you do not indicate otherwise, `LISTFILE` sends its output to `$STDLIST`. You can optionally specify a different output file to which the file descriptions are written.

LISTFILE examples

1. List all files in your logon account and group with file names that contain the characters "`INFO`":

```
LISTFILE @INFO@

INFOABST  INFOUTLN  INFOPREF  PSMGINFO  WINFOUTQ  XLINFO
```

2. Show the file characteristics of all files with names beginning with "X" in a specified account and group:

```
LISTFILE X@.INTRIN.LOZAR

ACCOUNT=  LOZAR          GROUP=    INTRIN

FILENAME    CODE ------------LOGICAL RECORD--- ----SPACE----

            SIZE  TYP     EOF     LIMIT R/B   SECTORS -X  MX
XLHPCICO     80B   FA      39       39   3         8   1   1
XLHPCIDE     80B   FA      47       47   3         8   1   1
XLHPCIGE     80B   FA      27       27   3         8   1   1
XLHPCIPU     80B   FA      44       44   3         8   1   1
```

3. Display the label information for a specified file:

```
LISTFILE ODDITY,3

********************
FILE: ODDITY.INFO.LOZAR

FILE CODE : 1030       FOPTIONS: BINARY,FIXED,NOCCTL,STD
BLK FACTOR: 1          CREATOR :
REC SIZE: 256(BYTES)   LOCKWORD:
BLK SIZE: 256(BYTES)   SECURITY--READ    : ANY
EXT SIZE: 0(SECT)                WRITE   : ANY
NUM REC: 7816                    APPEND  : ANY
NUM SEC: 0                       LOCK    : ANY
```

```
NUM EXT: 4                        EXECUTE : ANY
MAX REC: 31250                  **SECURITY IS ON
                      FLAGS    : n/a
NUM LABELS: 0         CREATED : TUE, JUN  3, 1986, 9:47 AM
MAX LABELS: 0         MODIFIED: TUE, JUN  3, 1986, 9:48 AM
DISC DEV -: 0         ACCESSED: WED, JUN  4, 1986, 2:38 PM
CLASS     : DISC      LABEL ADDR: $00000010 $00004414
SEC OFFSET: 0
```

4. Obtain a detailed ACD report on a file:

```
LISTFILE FILEA,-2

FILE = FILEA        ************** ACD ENTRIES **************

                    SAM.DOE              : R
                    JOE.DOE              : W
                    @.DESIGN             : NONE
                    @.@                  : X
```

In order to display information about such large files, two new file format options have been introduced. The LISTF, LISTFILE, and LISTFTEMP commands will now support options 10 and 11. File format option 10 is the large file equivalent of file option "1", containing all the same information, but in a new format that allows for the expression of larger values. Similarly, the format option 11 is the large file equivalent of file option "2". Again, the same basic information is displayed, but the format has been changed to allow for a greater range of values to be displayed.

The syntax of the LISTF, LISTFILE, and LISTFTEMP commands have not been changed with the addition of the new 10 and 11 format options. The same syntax can be used with the new values of 10 and 11 as the format. The output of the new formats will vary depending on whether or not MPE syntax names are being displayed or HFS syntax names are being displayed.

The format for each of MPE and HFS syntax varieties of the new formats 10 and 11 can be seen from the examples shown below

:listf @,10
ACCOUNT= SYS GROUP= EXAMPLE

Name	Access ERWS	FCode	RecSiz	Type	EOF	File Limit
BIGFILE			1024	FA	0	1023456789
ICE	E	NMPRG	256	FB	832	832
XKSM64			80	FAk	1	1023
XKSMXL			80	FAK	1	1023
XRAND	RW		80	FA	157	157
XRAND2	W		252	VA	386	49
YRAND		NMOBJ	256	FB	22	4000

:listfile ./@,10
PATH= /SYS/EXAMPLE/

Access ERWS	FCode	RecSiz	Type	EOF	File Limit	Name
		1024	FA	0	1023456789	BIGFILE
E	NMPRG	256	FB	832	832	ICE
		80	FAk	1	1023	XKSM64
		80	FAK	1	1023	XKSMXL
RW		80	FA	157	157	XRAND
W		252	VA	386	49	XRAND2
	NMOBJ	256	FB	22	4000	YRAND

:listf @,11
ACCOUNT= SYS GROUP= EXAMPLE

Name	Access ERWS	FCode	RecSiz	Type	EOF	File Limit	Disk Usage KB	Exts
BIGFILE			1024	FA	0	1023456789	0	0
ICE	E	NMPRG	256	FB	832	832	208	1
XKSM64			80	FAk	1	1023	64	1
XKSMXL			80	FAK	1	1023	52	1
XRAND	RW		80	FA	157	157	16	1
XRAND2	W		252	VA	386	49	16	1
YRAND		NMOBJ	256	FB	22	4000	8	1

listfile ./@,11
PATH= /SYS/EXAMPLE/

Access ERWS	FCode	RecSiz	Type	EOF	File Limit	Disk Usage KB	Exts	Name
		1024	FA	0	1023456789	0	0	BIGFILE
E	NMPRG	256	FB	832	832	208	1	ICE
		80	FAk	1	1023	64	1	XKSM64
		80	FAK	1	1023	52	1	XKSMXL
RW		80	FA	157	157	16	1	XRAND
W		252	VA	386	49	16	1	XRAND2

Access ERWS	FCode	RecSiz	Type	EOF	File Limit	Disk Usage KB	Exts	Name
	NMOBJ	256	FB	22	4000	8	1	YRAND

The display for the format options 10 and 11 is the same for each of the LISTF, LISTFILE, and LISTFTEMP commands. One of the unique additions of these two new options is the the "ERWS" column. This column indicates whether or not the file is currently opened, and if so, for what type of access. An "E" in the E column indicates that the file is opened for Execute access, the "R" in the R column indicates that the file is opened for Read access, the "W" in the W column indicates that the file is opened for Write access, and the "S" in the S column indicates that the file is being Stored by the STORE or TurboSTORE utility. The column may have one or more of these characters specified, or none, depending on how the file is currently accessed.

Displaying temporary file information with LISTFILE...(;TEMP)

The `LISTFILE...(;TEMP)` command is similar to `LISTFILE`, except that it displays information about the specified temporary files. Syntax, parameters, and information displayed are the same as `LISTFILE`, with the following exceptions:

Its display of file information includes the word `TEMPORARY` or the abbreviation `TEMP`.

Displaying file equations with LISTEQ

The `LISTEQ` command allows you to list all file equations in effect for the job or session from which you issue the command. Here is an example of a `LISTEQ` call:

```
LISTEQ


FILE EQUATIONS

FILE LP;DEV=PP;ENV=ELITE.XQENV.SYS;CCTL
FILE OFFLINE;DEV=PP;ENV=ELITE.XQENV.SYS;CCTL
FILE EDTLIST;DEV=PP;ENV=ELITE.ENV2680.SYS
```

If you do not indicate otherwise, `LISTEQ` sends its output to `$STDLIST`. You can optionally specify a different output file to which the file descriptions are written. For example, the following command sends output to the temporary file `MYFILE`:

```
LISTEQ MYFILE
```

Retrieving Specific File Information

The `LISTFILE`, `LISTFILE...;TEMP` and `LISTEQ` commands return formatted information to your job or session list device. If you need to retrieve specific information about a particular file, and you wish to place it in a variable available either to your CI or to your program, then you'll be interested in the the CI evaluator function and intrinsics described below.

You use:	To obtain information about:
`[CMD]FINFO,FFILEINFO,FGETINFO`	Characteristics of a currently opened file
`FLABELINFO`	Characteristics of a disk file (opened or not)
`FRELATE`	Whether files are interactive and/or duplicative

[:CMD] FINFO

You can use the FINFO evaluator function interactively to retrieve information about a specified file. FINFO is a function of the expression evaluator, a system procedure used by the `IF, WHILE, SETVAR,` and `CALC` commands of the command interpreter.

FINFO has two parameters. The first is the file name of the file about which you wish to obtain information; this is a string, and must be either a fully or partially qualified file name, or a `FILE` equation backreference. The second parameter is an integer or integer expression that indicates the nature of the information required. The options available are listed in Table 14-2. on page 194

Table 14-2. FINFO Options

Specify:	FINFO returns:
0	True if rhw file exists; False if it does not
1	Fully qualified file designator
4	Name of file creator
6	Date of file creation in format (*day,mmm,dd,yyyy*)
-6	Date of file creation in format (*yyyymmdd*)
8	Date of last modification in format (*day,mmm,dd,yyyy*)
-8	Date of last modification in format (*yyyymmdd*)
9	File code mnemonic or file code as string
-9	File code as integer
12	File limit
13	FOPTIONS (same format as `LISTFILE` -3)
-13	FOPTIONS

Table 14-2. FINFO Options

Specify:	FINFO returns:
14	Record size (negative value indicates bytes)
15	Block size
19	EOF marker location
24	Last modified time in format (*hh:mm am/pm*)
-24	Last modified time in format (*hhmmss*)
33	Lockword

For a more complete list of items that FINFO supports, type HELP FINFO at your terminal.

The example below shows the steps you can use to retrieve the following information about a file:

- the fully qualified file designator of the specified file
- the name of the file creator
- file characteristics, returned in the decimal, hexadecimal, and octal equivalents of the *foptions* format described in the FOPEN intrinsic description located in the *MPE/iX Intrinsics Reference Manual*.
- the file code, returned in decimal, hexadecimal, and octal equivalents

```
CALC FINFO ('MYFILE',1)
 MYFILE.MYGROUP.MYACCT
CALC FINFO ('MYFILE',4)
 SCOTT
CALC FINFO ('MYFILE',-13)
 1029, $405, %2005
CALC FINFO ('MYFILE',-9)
 0, $0, %0
```

FFILEINFO

Use this intrinsic to retrieve information about a specified file. The file can be on any device, but it must be opened by the calling process at the time of the FFILEINFO call. If you wish to return label information from a file that is not opened, use FLABELINFO instead. FFILEINFO has one required parameter, *filenum*. This is the file number, which is returned when you open a file using FOPEN or HPFOPEN. You can specify the information that you wish to be returned by using up to five *itemnum,item* pairs. Each *itemnum* designates a type of information (for example, logical device number, name of file creator, or volume ID), which is then returned in the *item* parameter. The *itemnum*s can be specified in any order.

Here is an example of an `FFILEINFO` intrinsic call. The information returned in this example is the same information retrieved in the`FINFO` example above:

```
HPFOPEN(FILENUM,STATUS);
FORMALDESIGNATOR:= EMPTYARRAY;
FILECODE:=0;
FOPTIONS:=0;
CREATOR:=EMPTYARRAY;
FFILEINFO(FILENUM,1,FORMALDESIGNATOR,18,CREATOR,2,FOPTIONS,8,FILECODE);
```

Here is a description of the information returned in the parameters specified in the above `FFILEINFO` call:

`FILENUM`	A variable of type 16-bit signed integer that returns the file number of the file about which information is requested.
`FORMALDESIGNATOR`	A variable of type character array that returns the actual file designator of the file, in the format *filename/groupname/accountname*.
`CREATOR`	A variable of type character array that returns the file creator name.
`FOPTIONS`	A variable of type 16-bit unsigned integer that returns file characteristics in the format described in the `FOPEN` intrinsic description.
`FILECODE`	A variable of type 16-bit signed integer that returns the file code.

A complete description of the information that you can obtain using `FFILEINFO` is given in the *MPE/iX Intrinsics Reference Manual*.

FGETINFO

This intrinsic, which returns some of the same information as `FFILEINFO`, is an MPE V/E-based intrinsic that is currently supported only for compatibility reasons. When you use a call to `FGETINFO`, MPE/iX now calls `FFILEINFO` to retrieve the file information. For this reason, it is advisable for you to call `FFILEINFO` directly; however, there is no need to rewrite existing programs that use `FGETINFO` unless there is a performance problem.

Here is an example of an `FGETINFO` intrinsic call that is the exact equivalent of the `FFILEINFO` example shown above:

```
HPFOPEN(FILENUM,STATUS);
FORMALDESIGNATOR:= EMPTYARRAY;
FILECODE:=0;
FOPTIONS:=0;
CREATOR:=EMPTYARRAY;

FGETINFO(FILENUM,FORMALDESIGNATOR,,FOPTIONS,,,,,FILECODE,,,,,,,,,,CREATOR);
```

A complete description of the information that you can obtain using `FGETINFO` is given in the *MPE/iX Intrinsics Reference Manual*.

FLABELINFO

The FLABELINFO intrinsic returns information from the file label of a disk file. The file need not be opened at the time of the intrinsic call. The information returned by this intrinsic is a subset of the information returned by FFILEINFO.

Here is an example of a FLABELINFO intrinsic call that returns the same information as the FFILEINFO and FGETINFO examples shown above:

```
FORMALDESIGNATOR:='MYFILE.MYGROUP.MYACCT ';
MODE:=0;
FSERRORCODE:=0;
ITEMNUMS[1]:=13;     {Bytes 1..2 return characteristics }
ITEMNUMS[2]:=9;      {Bytes 3..4 return file code       }
ITEMNUMS[3]:=1;      {Bytes 5..12 return file name      }
ITEMNUMS[4]:=2;      {Bytes 13..20 return group name    }
ITEMNUMS[5]:=3;      {Bytes 21..28 return account name  }
ITEMNUMS[6]:=4;      {Bytes 29..36 return creator name  }
ITEMNUMS[7]:=0;      {Zero indicates end of list        }
INITIALIZE_ITEMS;    {Procedure initializes ITEMS fields}
INITIALIZE_ITEMERRORS; {Procedure sets elements to zero }
FLABELINFO(FORMALDESIGNATOR,MODE,
           FSERRORCODE,ITEMNUMS,ITEMS,ITEMERRORS);
```

The ITEMS parameter above is a record structure, exactly 36 bytes in length, that can be declared in the following manner:

```
TYPE ITEMS_TYPE = RECORD
           FOPTIONS: 0..65565;    {2-byte unsigned integer}
           FILECODE: SHORTINT;    {2-byte signed integer  }
           FILENAME: PACKED ARRAY[1..8] OF CHAR;
           GROUPNAME: PACKED ARRAY[1..8] OF CHAR;
           ACCOUNTNAME: PACKED ARRAY[1..8] OF CHAR;
           CREATORNAME: PACKED ARRAY[1..8] OF CHAR;
       END;
```

A complete description of the information that you can obtain using FLABELINFO is given in the *MPE/iX Intrinsics Reference Manual*.

Determining Interactive/Duplicative Files with FRELATE

This intrinsic is used for one specific purpose: determining whether a pair of files (input file and list file) is interactive, duplicative, or both. **Interactive** means that the file requires human intervention for all input operations. For example, an input file and a list file opened to the same terminal would form an interactive pair. **Duplicative** means that all input to the input file is echoed automatically to the list file. For example, input to a keyboard is duplicated on the associated CRT.

The FRELATE intrinsic has two required parameters: *infilenum* is the file number of the input file, and *listfilenum* is the file number of the list file. Both of these numbers are returned when you create the files using HPFOPEN or FOPEN.

The intrinsic returns a 16-bit unsigned integer. If the files are an interactive pair, bit (15:1) =1, or =0 if they are not. If the files are a duplicative pair, bit (0:1) =1, or =0 if they are not.

A file can be interactive, duplicative, or both. These attributes do not change between the time the files are opened and the time they are closed. You can use FRELATE to obtain information about files on all devices.

Displaying File Error Information

Several file system intrinsics are designed specifically for handling errors. If an I/O error occurs, most file system intrinsics return a condition code indicating this.

You use:	To obtain information about:
FCHECK	File system intrinsic error number
FERRMSG	File system intrinsic error message
PRINTFILEINFO	File information error display

FCHECK

The FCHECK intrinsic returns an error code that indicates the nature of a file system I/O error. A table of error codes appears in the *MPE/iX Intrinsics Reference Manual*, or you can use FERRMSG (described below) to display an error message.

FCHECK has five optional parameters. The *filenum* parameter indicates the file for which error information is to be returned. If you set this parameter to zero, FCHECK assumes you want information about the *last* failed FOPEN call. The error code is returned in the *errorcode* parameter.

NOTE Do not use FCHECK to determine error conditions of you last failed HPFOPEN intrinsic call. Error conditions associated with HPFOPEN are returned in the HPFOPEN *status* parameter. Instead, you can use the HPERRMSG intrinsic to return a message explaining the nature of an HPFOPEN intrinsic error or warning.

Three other parameters give additional information about file system errors. The *tlog* parameter returns the number of half-words read or written if an I/O error occurs. The *blknum* parameter gives the logical record count for a spool file, or the physical record count for any other type of file. The *numrecs* parameter returns the number of logical records in the bad block.

You must use this intrinsic prior to calling FERRMSG, since the error code returned by FCHECK is used as a parameter in the call to FERRMSG.

FERRMSG

This intrinsic is used following a call to FCHECK, to return an error message explaining the nature of a file system error. It has three required parameters: *errorcode* is the error number returned by FCHECK, *msgbuf* returns the error message, and *msglgth* returns the length of the error message returned in *msgbuf*.

This example shows a call to FCLOSE. If this returns a CCL condition, a call to FCHECK requests the error code; then FERRMSG returns the error message associated with this code:

```
FCLOSE(FILENUM,1,0);
  IF CCODE = CCL
  THEN BEGIN
       FCHECK(FILENUM,ERRNUM);          {Returns error number    }
       FERRMSG(ERRNUM,MESSAGE,LENGTH);  {Returns error message   }
       PRINT(MESSAGE,-LENGTH,0);        {Prints error message to
                                          $STDLIST}
            TERMINATE;                  {Terminate process       }
            END;
```

If the FCHECK code has no assigned meaning, the following message is returned:

```
UNDEFINED ERROR errorcode
```

PRINTFILEINFO

This intrinsic prints a file information display on the job or session list device, $STDLIST. The information shown depends upon whether or not a file is opened when the error occurs. For files not yet opened, or for which the FOPEN intrinsic fails, the display is shown in Example 14-1.

Example 14-1. File Information Display, Unopened File

```
+-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
!  FILE NUMBER 5       IS UNDEFINED.              !   Line #1
!  ERROR NUMBER:   2    RESIDUE:   0       (WORDS) !   Line #2
!  BLOCK NUMBER:   0            NUMREC:   0        !   Line #3
+-------------------------------------------------+
```

The lines in this display show the following information:

Line #	Meaning
1	Warns that no corresponding file is open.
2	ERROR NUMBER indicates the last FOPEN error for the calling program. RESIDUE is the number of words not transferred in an I/O request; since no such request applies here, this is zero.
3	In this form, the BLOCK, NUMBER, and NUMREC fields are always zero.

For files that are open when a CCG (EOF error) or CCL (irrecoverable file error) was returned, the file information display appears as shown in Example 14-2.

Example 14-2. File Information Display, Opened File

```
+-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
!  FILE NAME IS TREEFILE.PSMG.LOZAR               !    Line #1
!  FOPTIONS:  NEW,ASCII,FORMAL,F,NOCCTL,FEQ,      !    Line #2
!             NOLABEL                             !    Line #3
!  AOPTIONS:  INPUT,NOMR,NOLOCK,DEF,BUF,NOMULTI,  !    Line #4
!             WAIT,NOCOPY                         !    Line #5
!  DEVICE TYPE:  0      DEVICE SUBTYPE:  9        !    Line #6
!  LDEV:  2       DRT:  4        UNIT:  1         !    Line #7
!  RECORD SIZE:  256    BLOCK SIZE:  256 (BYTES)  !    Line #8
!  EXTENT SIZE:  128    MAX EXTENTS:  8           !    Line #9
!  RECPTR:  0           RECLIMIT:   1023          !    Line #10
!  LOGCOUNT:  0             PHYSCOUNT:  0         !    Line #11
!  EOF AT:  0         LABEL ADDR:  %00201327630   !    Line #12
!  FILE CODE:  0    ID IS PAULA      ULABELS: 0   !    Line #13
!  PHYSICAL STATUS:  1000000000000001             !    Line #14
!  NUMBER WRITERS:  0    NUMBER READERS:  1       !    Line #15
!  ERROR NUMBER:  0       RESIDUE:  0             !    Line #16
!  BLOCK NUMBER:  0               NUMREC:  1      !    Line #17
+-------------------------------------------------+
```

The lines on the above display show the information listed in Table 14-3. on page 200

Table 14-3. PRINTFILEINFO Information

Line #	Meaning
1	The file name.
2,3	The *foptions* in effect.
4,5	The *aoptions* in effect.
6,7	The device type and subtype, logical device number, (LDEV), device reference table (DRT), and unit of the device on which the file resides. If the file is a spool file, the *ldev* is the virtual rather than the physical device.
8	The record and block size of the offending record, in bytes and words, as noted.
9	The size of the current extent and the maximum number of records in the file.
10	The current record pointer, and limit on number of records in the file.
11	The present count of logical and physical records.
12	The locations of the current EOF and header label of the file.
13	The file code, name of the file's creator, and number of user-created labels.
14	The physical (hardware) status of the device on which the file resides.

Table 14-3. PRINTFILEINFO Information

Line #	Meaning
15	NUMBER WRITERS is the number of FOPEN calls of the file with some type of WRITE access. NUMBER READERS is the number of FOPEN calls to the file with READ access. This field applies only to message files; it does not appear for other files.
16	The error number and residue.
17	The block number and number of records (NUMREC) for the file.

Writing a file system error-check procedure

Error checking intrinsics can be used throughout a program every time that there is an intrinsic call. Instead of repeating a call to PRINTFILEINFO many times, it is more efficient to write an error-check procedure and merely call this procedure where necessary.

The following example is a sample error-check procedure, named FILERROR. This procedure is declared at the beginning of the program; from that point on, it can be called with a single statement.

The procedure contains two parameters. FILENO is an identifier through which the file number is passed. The PRINTFILEINFO intrinsic then prints a file information display for that file. QUITNO is part of the abort message printed by the QUIT intrinsic. This enables you to determine the point at which the process was aborted.

```
PROCEDURE FILERROR(FILENO,QUITNO:SHORTINT);

   BEGIN
       PRINTFILEINFO(FILENO);
       QUIT(QUITNO);
   END;
```

A Pascal/XL Program Examples

The HP Pascal/XL program examples in this appendix are provided to help you better understand how to use MPE/iX file system intrinsics to perform common file access tasks.

Here is a short description of the task handled by each of the program examples in this appendix:

- **Program Example A-1** illustrates how you can open three different files–an unlabeled magnetic tape file, `$STDLIST`, and a new disk file–and copy records sequentially from the tape file to the disk file, while concurrently writing the records to `$STDLIST`.
- **Program Example A-2** illustrates how you can open a labeled magnetic tape file and a new disk file, print the user label to `$STDLIST`, then copy records sequentially from the tape file to the disk file. Play close attention to how the program closes the new disk file as a permanent file, and how it allows the user to specify alternate file designators if the file name already exists.
- **Program Example A-3** illustrates how you can use the sequential access method of reading records from an old disk file, then use the random access method of writing the records to a new labeled disk file.
- **Program Example A-4** illustrates how you can read from a file using random access method of data access. In addition, the program shows how you can use the `FREADSEEK` intrinsic to increase program performance by prefetching records, thus minimizing I/O wait-time.
- **Program Example A-5** illustrates how you can allow a user to update records in a shared data file. This program makes use of file locking intrinsics, `FLOCK` and `FUNLOCK`, to ensure exclusive access to the shared file during the update process.

Program example A-1

This program illustrates how you can open three different files–an unlabeled magnetic tape file, `$STDLIST`, and a new disk file–and copy records sequentially from the tape file to the disk file while concurrently writing the records to `$STDLIST`.

Program Algorithm

The task specified above is accomplished by following the steps described below. Also indicated are the intrinsics used to accomplish file access tasks and the name of the procedure where the task is accomplished:

1. Open (`HPFOPEN`) three files--an unlabeled magnetic tape file, and new disk file, and `$STDLIST` (see procedure `open_unlabeled_tape_file` and procedure `open_file`).
2. In a loop, sequentially read (`FREAD`) records from tape file, then write (`FWRITE`) them to both disk file and `$STDLIST` (see procedure `copy_tapefile_to_disk_file`). Continue loop till tape file's EOF is reached.

3. Close (`FCLOSE`) the tape file and the disk file (see `procedure close_file`).

If a file system intrinsic returns an unsuccessful condition code, procedure `handle_file_error` is called to print file information and then abort the program.

Source code listing

Example A-1. Sequential Access

```
$standard_level 'hp3000'$
$lines 100$
$code_offsets on$
$tables on$
$list_code on$
program open_close_example(input,output);

{***********************************************************************}
{                          DECLARATION PART                             }
{***********************************************************************}

const
   ccg            = 0;                {Condition code warning        }
   ccl            = 1;                {Condition code error          }
   cce            = 2;                {Condition code successful     }
   update         = 5;                {HPFOPEN item value            }
   save_temp      = 2;                {HPFOPEN item value            }
   save_perm      = 1;                {HPFOPEN item value            }
   new            = 0;                {HPFOPEN item value            }
   permanent      = 1;                {HPFOPEN item value            }
   write          = 1;                {HPFOPEN item value            }


type
   pac256         = packed array [1..256] of char;
   pac80          = packed array [1..80] of char;
   status_type    = record        {HPFOPEN status variable type}
                       case integer of
                          0 : (info    : shortint;
                               subsys  : shortint);
                          1 : (all     : integer);
                       end;

var
   disk_file      : integer;
   tape_file      : integer;
   filename       : pac80;
   std_list       : integer;
   std_in         : integer;
   outbuf         : pac80;

function  FREAD: shortint; intrinsic;    {Read from mag tape file        }
procedure HPFOPEN; intrinsic;            {Open tape, disk, $STDLIST files }
procedure FCLOSE; intrinsic;             {Close tape and disk files      }
procedure FWRITE; intrinsic;             {Write to disk and $STDLIST files}
procedure PRINTFILEINFO; intrinsic;      {If unsuccessful intrinsic call  }
procedure QUIT; intrinsic;               {If unsuccessful intrinsic call  }
```

```
procedure handle_file_error
          (
             file_num : shortint;
             quit_num : shortint
          );

{***********************************************************************}
{ procedure handle_file_error is invoked when a file system intrinsic   }
{ returns and unsuccessful condition code. File information is printed  }
{ to $STDLIST, then the program aborts.                                 }
{***********************************************************************}
begin
  PRINTFILEINFO (file_num);
  QUIT (quit_num);
end;                                    {end procedure                  }

procedure open_unlabeled_tape_file
          (
             var file_num : integer
          );

{***********************************************************************}
{ procedure open_unlabeled_tape_file opens a permanent unlabeled mag    }
{ tape file update access only.                                         }
{***********************************************************************}

const
                                        {**define HPFOPEN item numbers **}
   formal_designator_option   = 2;
   domain_option              = 3;
   access_type_option         = 11;
   device_class_optin         = 42;
   density_option             = 24;

var
                                        {**define HPFOPEN items       ** }
   file_name                  : pac80;
   permanent,update,density   : integer;
   device_class               : pac80;
   status                     : status_type;


begin
  file_name :='&tapefile&';
  permanent := 3;
  update    := 5;
  device_class := '&TAPE&';
  density      := 1600;
  HPFOPEN (file_num, status, formal_designator_option, file_name,
                             domain_option, permanent,
                             access_type_option, update
                             device_class_option, device_class
                             density_option, density  );

  if status.all <> 0 then
     handle_file_error (file_num, 1);
end;                                     {end procedure                  }
```

```
procedure open_file
          (
             var file_num  : integer;
                 file_name : pac80;
                 domain    : integer;
                 access    : integer
          );


{************************************************************************}
{ procedure open_file acts as a generic file open procedure allowing     }
{ you to specify the domain option and the access type option           }
{************************************************************************}


const
                                         {**define HPFOPEN item numbers**}
formal_designator_option  = 2;
   domain_option             = 3;
   access_type_option        = 11;
   ascii_binary_option       = 53;

var
   ascii          : integer;
                                         {**define scratch variables  ** }
   msgbuf         : pac80;
   status         : status_type;

begin
   ascii := 1;
   HPFOPEN (file_num, status, formal_designator_option, file_name,
                              domain_option, domain,
                              ascii_binary_option, ascii,
                              access_type_option, access);

   if status.all <> 0 then
      handle_file_error (file_num, 2);
end;                                      {end procedure                 }


procedure copy_tapefile_to_discfile
          (
             tape_file : integer;
             disk_file : integer
          );


{************************************************************************}
{ procedure copy_tapefile_to_discfile copies logical records            }
{ sequentially from tape file to disk file  with concurrent print to    }
{ stdlist.                                                              }
{************************************************************************}


var
```

```
   inbuf        : pac80;
   end_of_file  : boolean;
   read_length  : integer;
begin
  end_of_file := false;
  repeat
          {**In a loop, do a simple sequential read from tape file to ***}
          {**disk file.                                                **}

    read_length := FREAD (tape_file, inbuf, 80);
    if ccode = ccl then
       handle_file_error (tape_file, 3)
    else
      if ccode = ccg then
         end_of_file := true
      else
        begin
          FWRITE (std_list, inbuf, read_length, 0);
          if ccode <> cce then
             handle_file_error (std_list, 4);


          FWRITE (disk_file, inbuf, read_length,0);
          if ccode <> cce then
             handle_file_error (disk_file, 5);

        end
  until end_of_file;
end;                                      {end procedure                 }


procedure close_file
          (
             file_num : integer;
             disp     : integer
          );


{*************************************************************************}
{ procedure close_file is a generic file closing procedure that allows    }
{ you to specify the final disposition of the file.                       }
{*************************************************************************}

var
   msgbuf : pac80;

begin
  FCLOSE (file_num, disp, 0);
  if ccode = ccl then
     handle_file_error (file_num, 6);
end;                                      {end procedure                 }


{*************************************************************************}
{                             MAIN PROGRAM                                }
{*************************************************************************}
```

```
begin
  open_unlabeled_tape_file (tape_file);               { STEP 1           }
  filename := '&$stdlist&';                           { STEP 1           }
  open_file (std_list, filename, permanent,write);    { STEP 1           }
  filename := '&dataone&';                            { STEP 1           }
  open_file (disk_file, filename, new,update);        { STEP 1           }
  copy_tapefile_to_discfile(tape_file,disk_file);     { STEP 2           }
  close_file(disk_file, save_temp);                   { STEP 3           }
  close_file(tape_file, save_perm);                   { STEP 3           }
end.                                             {end program            }
```

Program Example A-2

This Pascal/XL program example illustrates how you can use the HPFOPEN intrinsic to open a labeled magnetic tape file, then open a new disk file with a user-supplied name. After records are sequentially copied from the tape file to the disk file, both files are closed, the disk file is closed as a Permanent file. If the file system determines that another file of the same name exists in the permanent file directory, the user is allowed to specify alternate file names until the file close operation is successful.

Program Algorithm

The task specified above is accomplished using six steps. Also indicated are the intrinsics used to accomplish file access tasks and the name of the procedure where the task is accomplished:

1. Open (HPFOPEN) labeled magnetic tape file (see procedure open_tape_file).
2. Read from $STDIN (READ) a user-supplied file name, then open (HPFOPEN) a new disk file using the given name (see procedure open_disk_file).
3. Read (FREADLABEL) the user label from the tape file and then print (PRINT) the label to $STDLIST (see procedure print_user_label).
4. In a loop, use sequential access method to read (FREAD) records from tape file and write (FWRITE) them to the disk file (see procedure copy_file_from_tape_to_disc).
5. Close (FCLOSE) the tape file (see procedure close_tape_file).
6. Close (FCLOSE) the new disk file as a permanent file (see procedure close_disk_file). If an error occurs during the FCLOSE call, the user is given the opportunity (CAUSEBREAK) to interactively fix the problem (see procedure handle_fclose_error) before the program again attempts to close the disk file as a permanent file.

This program makes extensive use of error handling routines to:

- return to the user a file system error number (FCHECK) associated with a file system intrinsic error (refer to procedure print_fserr).
- interpret and return to the user error information returned by the status parameter of a failed HPFOPEN call (see procedure print_hpfopen_error).

- allow the user to specify an alternative file name if, during an `FCLOSE` call, the file system determines that a duplicate permanent disk file exists (see procedure `handle_fclose_error`).
- print file information (`PRINTFILEINFO`) before aborting (`QUIT`) the program (see procedure `handle_file_error`).

Using these four error procedures, the program individually tailors error-handling routines to meet different intrinsic needs.

Source code listing

Example A-2. Accessing a Magnetic Tape File

```
$standard_level 'os_features'$
$os 'mpe xl'$
$code_offsets on$
$tables on$
$list_code on$

program open_and_read_a_labeled_tape (input, output);


{*****************************************************************}
{*                        DECLARATION PART                       *}
{*****************************************************************}



const
   ccg              = 0;            {* condition code "greater than *}
   ccl              = 1;            {* condition code "less than"   *}
   cce              = 2;            {* condition code "equal"       *}

type
   pac80            = packed array [1..80] of char;
   status_type      = record
                         case integer of
                            0 : (info    : shortint;
                                 subsys  : shortint);
                            1 : (all     : integer);
                         end;

var
   tape_file        : integer;       {* file number for tape file    *}
   disk_file        : integer;       {* file number for disk file    *}

function  FREAD : shortint;  intrinsic;
function  READ  : shortint;  intrinsic;
procedure HPFOPEN;           intrinsic;
procedure FCHECK;            intrinsic;
procedure FCLOSE;            intrinsic;
procedure FWRITE;            intrinsic;
procedure PRINT;             intrinsic;
procedure PRINTFILEINFO;     intrinsic;
procedure QUIT;              intrinsic;
procedure CAUSEBREAK;        intrinsic;
```

```
procedure FREADLABEL;        intrinsic;

procedure print_hpfopen_error
          (
               error    : status_type
          )
option inline;

{**********************************************************************}
{*  PURPOSE:                                                          *}
{*    This routine prints the status returned by HPFOPEN.             *}
{*  PARAMETERS:                                                       *}
{*    error (input)                                                   *}
{*      - status returned by HPFOPEN                                  *}
{**********************************************************************}

begin                                            {* print_hpfopen_error *}
  writeln ('HPFOPEN status = (info: ', error.info:1,
                             '; subys: ', error.subsys:1,')');
end;                                             {* print_hpfopen_error *}


procedure print_fserr
          (
              file_num  : integer
          )
option inline;


{**********************************************************************}
{*  PURPOSE:                                                          *}
{*    This routine prints a File System error which occurred in a     *}
{*    File System intrinsic.                                          *}
{*  PARAMETERS:                                                       *}
{*    file_num (input)                                                *}
{*      - file number of file which the intrinsic failed              *}
{**********************************************************************}


var
   error              : shortint;          {* File System error number   *}

begin                                                    {* print_fserr *}
  FCHECK (file_num, error);        {* call FCHECK to get the errornumber*}
  writeln ('FSERR = ', error:1);
end;
                                                         {* print_fserr *}


procedure handle_file_error
          (
               file_num : shortint;
               quit_num : shortint
          );

{**********************************************************************}
{*  PURPOSE:                                                          *}
```

```
{*    This routine displays File System information about a file       *}
{*    and then calls QUIT to terminate the program.                    *}
{*  PARAMETERS:                                                        *}
{*    file_num (input)                                                 *}
{*      - file number.  The routine will print info about this         *}
{*        file.                                                        *}
{*    quit_num (input)                                                 *}
{*      - quit number.  This number will be displayed by QUIT when     *}
{*        the program is terminated.                                   *}
{**********************************************************************}


begin                                                 {* handle_file_error *}
  PRINTFILEINFO (file_num);
  QUIT (quit_num);
end;                                                  {* handle_file_error *}

procedure handle_fclose_error;

{**********************************************************************}
{*  PURPOSE:                                                           *}
{*    This routine informs the user that the disk file could not       *}
{*    closed.  Then CAUSEBREAK is called to break the program;         *}
{*    this is done to give the user a chance to purge or rename        *}
{*    an existing disk file which has the same name as the one the     *}
{*    program is trying to save.  When the user enters 'resume'        *}
{*    this routine will return to the caller.                          *}
{**********************************************************************}

var
   msgbuf            : pac80;

begin                                               {* handle_fclose_error *}
                                                   {* print error messages *}
                                                   {************************}

  msgbuf := 'Can''t close disk file';
  PRINT (msgbuf, -21, 0);
  msgbuf := 'Check for duplicate name';
  PRINT (msgbuf, -24, 0);
  msgbuf := 'Fix, then type "resume"';
  PRINT (msgbuf, -23, 0);

                                                      {* break the program *}
                                                      {*********************}
  CAUSEBREAK;
end;                                                {* handle_fclose_error *}


procedure open_tape_file
          (
           var file_num : integer
          );

{**********************************************************************}
{*  PURPOSE:                                                           *}
{*    This routine opens a labeled tape file.                          *}
{*  PARAMETERS:                                                        *}
```

```
{*    file_num (output)                                                *}
{*      - file number of open tape file                                *}
{**********************************************************************}

const
                                         {* define HPFOPEN item numbers *}
   formal_designator_option =  2;
   domain_option            =  3;
   tape_label_option        =  8;
   access_type_option       = 11;
   tape_type_option         = 30;
   tape_expiration_option   = 31;
   device_class_option      = 42;

var
                                                {* define HPFOPEN items *}
   read_only          : integer;
   device_class       : pac80;
   old                : integer;
   file_name          : pac80;
   tape_label         : pac80;
   ansi_tape          : integer;
   tape_expiration    : pac80;

   {* define scratch varibles *}
   msgbuf             : pac80;
   status             : status_type;

begin                                                  {* open_tape_file *}
                    {* set up the item values for the HPFOPEN intrinsic *}
                    {***************************************************}
  file_name        := '&tapefile&';
  old              := 3;
  read_only        := 0;
  tape_label       := '&tape01&';
  ansi_tape        := 0;
  tape_expiration  := '&05/20/87&';
  device_class     := '&tape&';
  HPFOPEN (file_num, status, formal_designator_option, file_name,
                         device_class_option, device_class,
                         domain_option, old,
                         tape_label_option, tape_label,
                         tape_type_option, ansi_tape,
                         access_type_option, read_only,
                         tape_expiration_option, tape_expiration);

  if status.all <> 0 then                   {* check for error condition *}
   begin
     print_hpfopen_error (status);
     handle_file_error (file_num, 1);
   end;
end;                                                   {* open_tape_file *}


procedure open_disk_file
          (
           var file_num : integer
          );
```

```
{**********************************************************************}
{*  PURPOSE:                                                          *}
{*    This routine prompts the user for a file name and opens a       *}
{*    NEW disk file using the given name.                             *}
{*  PARAMETERS:                                                       *}
{*    file_num (output)                                               *}
{*      - file number of the open disk file                           *}
{**********************************************************************}

const
                                        {* define HPFOPEN item numbers *}
   formal_designator_option =  2;
   access_type_option       = 11;
   ascii_binary_option      = 53;

var
                                               {* define HPFOPEN items *}
   update             : integer;
   ascii              : integer;
   file_name          : pac80;

                                           {* define scratch variables *}
   index              : integer;
   msgbuf             : pac80;
   read_length        : integer;
   status             : status_type;

begin                                             {* open_disk_file    *}
         {* prompt user for a file name a read the user-specified name  *}
         {*************************************************************}

  msgbuf := 'Name of new disk file to be created?';
  PRINT (msgbuf, -36, 0);

  read_length := READ (file_name, -8);

{* shift file name one character to the right to make room for the    *}
{* delimiters                                                          *}
{**********************************************************************}

  for index := read_length downto 1 do
     file_name[index + 1] := file_name[index];

                                        {* add delimiters to file name *}
                                        {******************************}

  file_name[1] := '&';
  file_name[read_length + 2] := '&';
          {* set up the remaining item values for the HPFOPEN intrinsic *}
          {************************************************************}

  ascii  := 1;                  {* the disk file is to be an ASCII file *}
  update := 5;   {* update access will be used to write to the disk file*}

  HPFOPEN (file_num, status, formal_designator_option, file_name,
                         ascii_binary_option, ascii,
                         access_type_option, update);
```

```
  if status.all <> 0 then                    {* check for error condition *}
     begin
       print_hpfopen_error (status);
       handle_file_error (file_num, 2);
     end;
end;                                        {* open_disk_file  *           }


procedure print_user_label
          (
               file_num : integer
          );

{************************************************************************}
{*  PURPOSE:                                                            *}
{*    This routine reads the user label from the tape file and         *}
{*    then prints the user label to $STDLIST.                           *}
{*  PARAMETERS:                                                         *}
{*    file_num (input)                                                  *}
{*      - file number of open tape file                                 *}
{************************************************************************}

var
   inbuf             : pac80;                {* buffer for the user label *}

begin   {* print_user_label *}
  FREADLABEL (file_num, inbuf, 40);       {* read the user label from tape*}

  if ccode <> CCE then                      {* check for error condition *}
    begin
      print_fserr (file_num);
      handle_file_error (file_num, 3);
    end;

  PRINT (inbuf, 40, 0);               {* print the user label to $stdlist *}
end;                                                {* print_user_label *}


procedure copy_file_from_tape_to_disk
          (
               tape_file : integer;
               disk_file : integer
          );
{************************************************************************}
{*  PURPOSE:                                                            *}
{*   This routine copies a tape file to a disk file one record at      *}
{*   a time (sequential access).                                        *}
{*  PARAMETERS:                                                         *}
{*    tape_file (input)                                                 *}
{*      - file number of an open tape file                              *}
{*    disk_file (input)                                                 *}
{*      - file number of an open disk file                              *}
{************************************************************************}


var
```

```
   inbuf              : pac80;
   msgbuf             : pac80;
   end_of_file        : boolean;
   read_length        : integer;

begin                                    {* copy_file_from_tape_to_disk *}
  end_of_file := false;

  repeat
   {* copy a buffer from the tape file to the disk file until the      *}
   {* end of the tape file is reached                                  *}
   {*******************************************************************}

    read_length := FREAD (tape_file, inbuf, 40);
                                              {* read buffer from tape  *}

    if ccode = ccl then                    {* check for error condition *}

      begin
        msgbuf := 'Can''t read tape file';
        PRINT (msgbuf, -20, 0);
        print_fserr (tape_file);
        handle_file_error (tape_file, 4);
      end
    else
    if ccode = ccg then        {* check for end of file condition *}
      end_of_file := true
    else
      begin
        FWRITE (disk_file, inbuf, read_length, 0);
                                              {* write buffer to disk *}
        if ccode <> cce then   {* check for error condition *}
          begin
            msgbuf := 'Can''t write to disk file';
            PRINT (msgbuf, -24, 0);
            print_fserr (disk_file);
            handle_file_error (disk_file, 5);
         end;
      end;
  until end_of_file;
end;                                     {* copy_file_from_tape_to_disk *}

procedure close_tape_file
          (
               file_num : integer
          );

{*********************************************************************}
{*  PURPOSE:                                                         *}
{*    This routine closes the tape file.                             *}
{*  PARAMETERS:                                                      *}
{*    file_num (input)                                               *}
{*      - file number of open tape file                              *}
{*********************************************************************}


var
```

```
    msgbuf            : pac80;

begin                                           {* close_tape_file *}
  FCLOSE (file_num, 1, 0);          {* close file, rewind and unload tape*}
  if ccode = ccl then                       {* check for error condition *}
     begin
       msgbuf := 'Can''t close tape file';
       PRINT (msgbuf, -21, 0);
       print_fserr (file_num);
       handle_file_error (file_num, 6);
     end;
end;                                    { close_tape_file              }


procedure close_disk_file
          (
               file_num : integer
          );

{*************************************************************************}
{*  PURPOSE:                                                              }
{*    This routine closes the NEW disk file as PERMANENT disk             }
{*    file.  If an error occurs on the FCLOSE then the user is            }
{*    given the opportunity to fix the problem and the FCLOSE is          }
{*    retried.                                                            }
{*  PARAMETERS:                                                           }
{*    file_num (input)                                                    }
{*      - file number of the open disk file                               }
{*************************************************************************}

var
   file_closed       : boolean;

begin                                   { close_disk_file              }
  file_closed := false;
  repeat
    FCLOSE (file_num, 1, 0);        { close disk file as a permanent file}

    if ccode = ccl then                     { check for error condition}
      handle_fclose_error
    else
      file_closed := true;
  until file_closed;
end;                                    { close_disk_file              }


{****************************************************************************}
   {                          MAIN PROGRAM                                  }

{****************************************************************************}

   begin
     open_tape_file (tape_file);                         { STEP 1       }
     open_disk_file (disk_file);                         { STEP 2        }
     print_user_label (tape_file);                       { STEP 3        }
     copy_file_from_tape_to_disk (tape_file, disk_file);  { STEP 4
```

```
}
     close_tape_file (tape_file);                             { STEP 5         }
     close_disk_file (disk_file);                             { STEP 6         }

   end.                                                      {     main       }
```

Program Example A-3

This HP Pascal/XL program illustrates how you can use a sequential method of reading records from an old disk file and use a random access method of writing the records in an inverted order to a new user-labeled disk file, where record 1 of the first file is written to location n of the second file, record 2 is written to location *n*-1, and so on.

Program Algorithm

The task specified above is accomplished by following the steps described below. Also indicated are the intrinsics used to accomplish file access tasks and the name of the procedure where the task is accomplished:

1. Open (`HPFOPEN`) a permanent disk file and a new user-labeled disk file (see procedure `open_disk_file`).
2. Write (`FWRITELABEL`) a user-defined label to the new file (see procedure `write_user_label`).
3. Get EOF (`FGETINFO`) of old file and assign that value to new file's record pointer; in a loop, sequentially read (`FREAD`) records from old file and write (`FWRITEDIR`) them to a location in the new file specified by the record pointer, then decrement the new file's record pointer (see procedure `copy_oldfile_to_newfile`). Continue the loop till the old file's EOF is reached.
4. Close (`FCLOSE`) the old file as deleted from the system, and close the new file as a temporary file (see procedure `close_disk_file`).

If a file system intrinsic returns an unsuccessful condition code, procedure `handle_file_error` is called to print file information (`PRINTFILEINFO`) and then abort (`QUIT`) the program.

Source code listing

Example A-3. Random Access

```
$standard_level 'hp3000'$
$lines 100$
$code_offsets on$
$tables on$
$list_code on$
program write_read (input,output);

{*********************************************************************}
```

```
{                         DECLARATION PART                                  }
{***************************************************************************}

const
   ccg              = 0;           { condition code warning/EOF,/etc.. }
   ccl              = 1;           { condition code error              }
   cce              = 2;           { condition code successful         }
   permanent        = 1;
   new              = 0;
   temp             = 2;
   delete           = 4;

type
   pac256           = packed array [1..256] of char;
   pac80            = packed array [1..80] of char;
                                   {HPFOPEN status parameter type     }
   status_type      = record
                        case integer of
                           0 : (info     : shortint;
                                subsys   : shortint);
                           1 : (all      : integer);
                        end;

var
   old_file       : integer;
   new_file       : integer;
   filename       : pac80;
   label_id       : integer;
   label_len      : integer;
   outbuf         : pac80;

function FREAD: shortint; intrinsic;  { sequential read old file         }
procedure HPFOPEN; intrinsic;         { open both disk files             }
procedure FCLOSE; intrinsic;          { close both disk files            }
procedure FWRITEDIR; intrinsic;       { random access write to new file  }
procedure FWRITELABEL; intrinsic;     { write new user-defined label     }
procedure PRINTFILEINFO; intrinsic;   { user in error-handler            }
procedure FGETINFO; intrinsic;        { get EOF location                 }
procedure QUIT; intrinsic;            { use in error-handler             }

procedure handle_file_error
          (
             file_num : shortint;
             quit_num : shortint
          );
{***************************************************************************}
{ procedure handle_file_error prints file information on the job/session}
{ list device, then aborts the program.                                 }
{***************************************************************************}

begin
  PRINTFILEINFO (file_num);
  QUIT (quit_num);
end;                               { end handle_file_error             }


procedure open_disk_file
          (
```

```
             var file_num  : integer;
                 file_name : pac80;
                 domain    : integer
          );

{*********************************************************************}
{procedure open_disk_file is a generic file open procedure that allows }
{you to specify the file name, it's domain, type of access, and internal}
{format - ASCII or binary.                                              }
{*********************************************************************}

const
                                      {**define HPFOPEN item numbers**   }
   formal_designator_option  = 2;
   domain_option             = 3;
   access_type_option        = 11;
   ascii_binary_option       = 53;

var
                                      {**define HPFOPEN items*********   }
   update         : integer;
   ascii          : integer;

                                      {**define scratch variables**      }

   msgbuf         : pac80;
   status         : status_type;

begin
   update := 5;
   ascii := 1;

   HPFOPEN (file_num, status, formal_designator_option, file_name,
                              domain_option, domain,
                              ascii_binary_option, ascii,
                              access_type_option, update);

   if status.all <> 0 then
      handle_file_error (file_num, 1);
end;                                  { end open_disk_file               }

procedure write_user_label
          (
             file_num : integer;
             buffer   : pac80;
             length   : integer;
             lnum     : integer
          );

{**********************************************************************}
{ procedure write_user_label writes a user-defined label to the specified}
{ file.                                                                  }
{**********************************************************************}

begin
  FWRITELABEL (file_num, buffer, length, lnum);
  if ccode <> cce then
    handle_file_error (file_num, 2);
```

```
end;                                 { end write_user_label              }

procedure copy_oldfile_to_newfile
          (
             new_discfile : integer;
             old_discfile : integer
          );

{*********************************************************************}
{ procedure copy_oldfile_to_newfile gets EOF of old file & assigns record}
{ pointer to that value. In a loop, sequentially reads from old file;   }
{ random access writes to new file.                                     }
{*********************************************************************}

var
   rec          : integer;
   inbuf        : pac256;
   end_of_file  : boolean;
   read_length  : integer;

begin

                                     {**Locate the EOF in old disk file** }
  end_of_file := false;              {  initialize loop control variable  }
  rec := 0;

  FGETINFO (old_discfile,,,,,,,,,, rec);
  if ccode = ccl then
    handle_file_error (old_discfile, 3);

  repeat
           {**Copy the records in the reverse orders from old disk file**}
                    {**to the new disk file**                            }

   read_length := FREAD (old_discfile, inbuf, 128);
   if ccode = ccl then
      handle_file_error (old_discfile, 4)
       else
      if ccode = ccg then
         end_of_file := true
      else
begin
          rec := rec - 1;             { decrement record pointer          }
          FWRITEDIR (new_discfile, inbuf, 128, rec);
          if ccode <> cce then
            handle_file_error (new_discfile, 5);
        end
  until end_of_file                   { check control variable EOF        }
end;                                  { end copy_oldfile_to_newfile       }


procedure close_disk_file
          (
             file_num : integer;
             disp     : integer
          );
```

```
{*********************************************************************}
{ procedure close_disk_file is a disk file closing procedure that allowsa }
{ you to specify the final disposition of the file you are closing.      }
{*********************************************************************}

var
   msgbuf : pac80;

begin
  FCLOSE (file_num, disp, 0);
  if ccode = ccl then
     handle_file_error (file_num, 6);
end;                                    { end close_disk_file          }


{*********************************************************************}
{                          Main Program                                }
{*********************************************************************}


begin
filename := '&dataone&';
open_disk_file (old_file, filename, permanent);          { STEP 1       }
filename := '&datatwo&';
open_disk_file (new_file, filename, new);                { STEP 1       }
outbuf := 'Employee Data File';
label_len := 9;
label_id := 0;
write_user_label(new_file, outbuf, label_len, label_id); { STEP 2       }
copy_oldfile_to_newfile(new_file, old_file);             { STEP 3       }
close_disk_file(new_file, temp);                         { STEP 4       }
close_disk_file (old_file, delete);                      { STEP 4       }
end.
```

Program Example A-4

This HP Pascal/XL program illustrates how you can use the FREADSEEK intrinsic to improve I/O performance during random access reads. The program opens a permanent disk file containing data, and $STDLIST. Even numbered records are read from the data file and printed to $STDLIST.

Program Algorithm

The task specified above is accomplished by following the steps described below. Also indicated are the intrinsics used to accomplish file access tasks and the name of the procedure where the task is accomplished:

1. Open (FOPEN) both the the Permanent disk file and $STDLIST (see procedure open_files).
2. Read (FREADLABEL) the user label from the disk file and write (FWRITE) it to $STDLIST

(see procedure `read_user_label`).

3. In a loop, read (`FREADDIR`) even numbered records from the disk file. Before writing (`FWRITE`) the records to disk, prefetch the next record (`FREADSEEK`). Do this till EOF of the disk file is reached (see procedure `read_from_datafile`).
4. Close (`FCLOSE`) both files (see procedure `close_files`).

If a file system intrinsic returns an unsuccessful condition code, procedure `handle_file_error` is called to print file information (`PRINTFILEINFO`) and then abort (`QUIT`) the program.

Source code listing

Example A-4. Random Access

```
program Read_Example (input,output);
{**********************************************************************}
{                         DECLARATION PART                             }
{**********************************************************************}

  const
    CCG = 0;                             { condition code warning      }
    CCL = 1;                             { condition code error        }
    CCE = 2;                             { condition code successful   }
  type
    file_name = packed array [1..9] of char;
    buffertype = packed array [1..80] of char;


  var
    datafile_name: file_name;
    listfile_name: file_name;
    buffer       : buffertype;
    message      : buffertype;
    datafile     : shortint;
    listfile     : shortint;
    record_num   : integer;

  function fopen:shortint; intrinsic;    { open files                  }
  procedure freadlabel;    intrinsic;    { read user-defined label     }
  procedure freaddir;      intrinsic;    { random access read file     }
  procedure fwrite;        intrinsic;    { sequential write to $STDLIST }
  procedure fclose;        intrinsic;    { close files                 }
  procedure freadseek;     intrinsic;    { prefetch selected record    }
  procedure printfileinfo; intrinsic;    { used in error-handler       }
  procedure quit;          intrinsic;    { used in error-handler       }



  procedure error_handler (filenum, quitnum: shortint);
{**********************************************************************}
{ procedure error_handler is a standard file system error handling     }
{ procedure invoked after an unsuccessful file system intrinsic call.  }
{ A file information display is printed to $STDLIST, then program aborts.}
{**********************************************************************}
```

```
    begin
      printfileinfo (filenum);
      quit (quitnum);
    end;                                  {end error_handler              }

  procedure open_files;
{***********************************************************************}
{ procedure open_files opens the data file and $STDLIST using the FOPEN  }
{ intrinsic.                                                            }
{***********************************************************************}

    const
      permanent     = 5;
      read_write    = 4;
      stdlist      = 12;
      write        =  1;

    begin
      datafile_name:= 'datafile ';
      listfile_name:= 'listfile ';
      datafile:= fopen(datafile_name,permanent,read_write,-80);
      if ccode <> CCE then error_handler(datafile,1);
      listfile:= fopen(listfile_name,stdlist,write);
      if ccode <> CCE then error_handler(listfile,2);
    end;                                        {end open_files           }


procedure read_user_label;
{***********************************************************************}
{procedure read_user_label reads the user label located in the          }
{user-defined label portion of the data file, then prints it to $STDLIST.}
{***********************************************************************}

  begin
    freadlabel(datafile,buffer,-80);
    if ccode <> CCE then error_handler(datafile,101);
    fwrite (listfile,buffer,-80,0);
    if ccode <> CCE then error_handler(listfile,102);
  end;                                          {end read_user_label      }

procedure read_from_datafile;
{***********************************************************************}
{ procedure read_from_data_file first calls procedure read_user_label to }
{ print the label to $STDLIST, then enters a loop to select only even   }
{ numbered records from the data file and writing them to $STDLIST.     }
{***********************************************************************}

  var end_of_file: boolean;

  begin

    end_of_file:= false;                        {initialize loop control  }
    record_num:= 0;


                                    { enter loop, random access read even }
                                    { numbered record, freadseek next     }
```

```
                                      { selection, then sequential write     }
                                      { to $STDLIST, till EOF.               }

    while not end_of_file do
      begin
        freaddir(datafile,buffer,-80,record_num);
        if ccode <> CCE then error_handler(datafile,103);
        record_num:= record_num + 2;
        freadseek(datafile,record_num);
        if ccode = CCL then error_handler(datafile,104) else
        if ccode = CCG then end_of_file:= true;
        fwrite(listfile,buffer,-80,0);
        if ccode <> CCE then error_handler(listfile,105);
      end;
  end;                                           {end read_from_datafile   }



procedure close_files;
{************************************************************************}
{ procedure close_files  calls the fclose intrinsic twice to close bot   }
{ files previously opened by procedure open_files.                       }
{************************************************************************}


  begin
    fclose(datafile,0,0);
    if ccode <> CCE then error_handler(datafile,1001);
    fclose(listfile,0,0);
    if ccode <> CCE then error_handler(listfile,1002);
  end;



{************************************************************************}
{                        MAIN PROGRAM                                    }
{************************************************************************}

begin
  open_files;                                           { STEP 1           }
  read_user_label;                                      { STEP 2           }
  read_from_datafile;                                   { STEP 3           }
  close_files;                                          { STEP 4           }

end.                                                    {end main program   }
```

Program Example A-5

This HP Pascal/XL program example illustrates how you can update a particular record of a shared data file. In addition, this program example uses file system locking intrinsics (FLOCK, FUNLOCK) to ensure exclusive access to the file while the update occurs.

Program Algorithm

The task specified above is accomplished by following the steps described below. Also indicated are the intrinsics used to accomplish file access tasks and the name of the procedure where the task is accomplished:

1. Open (HPFOPEN) three files, $STDLIST, $STDIN, and a permanent disk file containing data to update (see procedure open_file).
2. In a loop, lock (FLOCK) a shared data file; read (FREAD) data from disk file; write (FWRITE) data to $STDLIST; read (FREAD) new data from $STDIN; update (FUPDATE) shared data file with data read from $STDIN. The loop ends when EOF of disk file is reached (see procedure update_file).
3. Close (FCLOSE) the disk file (see procedure close_disk_file); let normal program termination close the other files.

If a file system intrinsic returns an unsuccessful condition code, procedure handle_file_error is called to print file information (PRINTFILEINFO) and then abort (QUIT) the program.

Source code listing

Example A-5. Updating a Shared File

```
$standard_level 'hp3000'$
$lines 100$
$code_offsets on$
$tables on$
$list_code on$
program access_file3(input,output);

{************************************************************************}
{                          DECLARATION PART                              }
{************************************************************************}

const
   ccg              = 0;                { condition code warning       }
   ccl              = 1;                { condition code warning       }
   cce              = 2;                { condition code successful    }
{ HPFOPEN item values}
   permanent        = 1;
   read             = 0;
   write            = 1;
   update           = 5;
   save             = 1;
   shared           = 4;
   locking          = 1;

type
   pac256           = packed array [1..256] of char;
   pac80            = packed array [1..80] of char;

                                        { HPFOPEN status type          }
   status_type      = record
                         case integer of
                            0 : (info     : shortint;
```

```
                                subsys   : shortint);
                           1 : (all      : integer);
                        end;

   var
      disk_file      : integer;
      filename       : pac80;
      std_list       : integer;
      std_in         : integer;
      outbuf         : pac80;

   function FREAD: shortint; intrinsic;       { sequential reads              }
   procedure HPFOPEN; intrinsic;              { open files                    }
   procedure FCLOSE; intrinsic;               { close files                   }
   procedure FWRITE; intrinsic;               { sequential writes             }
   procedure FWRITEDIR; intrinsic;            { random access writes          }
   procedure FUNLOCK; intrinsic;              { unlock locked file            }
   procedure PRINTFILEINFO; intrinsic;        { use in error handler          }
   procedure FLOCK; intrinsic;                { lock file                     }
   procedure FUPDATE; intrinsic;              { update record                 }
   procedure QUIT; intrinsic;                 { use in error handler          }
   procedure handle_file_error
             (
                file_num : shortint;
                quit_num : shortint
             );


{*****************************************************************}
   { procedure handle_file_errorPrints the file information on the
}
   { session/job list device.                                    }

{*****************************************************************}

   begin
     PRINTFILEINFO (file_num);
     QUIT (quit_num);
   end;                                 { end handle_file_error       }
   procedure open_file
             (
                var file_num  : integer;
                    file_name : pac80;
                    domain    : integer;
                    access    : integer;
                    excl      : integer
                    lockable  : integer;
             );


{*****************************************************************}
   { procedure open_file is a generic file opening procedure that allows
you}
   { to specify the designator, domain, access type, ASCII/binary, and
```

```
}
   { exclusive options for the file.                                        }
{************************************************************************}

   const
   {**define HPFOPEN item numbers**}
      formal_designator_option  = 2;
      domain_option             = 3;
      access_type_option        = 11;
      ascii_binary_option       = 53;
      exclusive_option          = 13;
      dynamic_locking_option    = 12;


   var
      ascii          : integer;

                                              {define scratch variables      }

      msgbuf         : pac80;
      status         : status_type;

   begin
      ascii := 1;

      HPFOPEN (file_num, status, formal_designator_option, file_name,
                                 domain_option, domain,
                                 ascii_binary_option, ascii,
                                 access_type_option, access,
                                 exclusive_option, excl
                                 dynamic_locking_option, lockable);
      if status.all <> 0 then
         handle_file_error (file_num, 1);
   end;                                       { end open_file                }

   procedure update_file
             (
                old_discfile : integer
             );

   {************************************************************************}
   { procedure update_file pdates records in the disk file with the         }
   { replacement read from the stdin.                                       }
   *************************************************************************}

   var
      dummy        : integer;
      inbuf        : array [1..80] of char;
      end_of_file  : boolean;
      read_length  : integer;

   begin
                                              {Lock the file and suspend     }
```

```
    end_of_file := false;
    FLOCK (old_discfile,1);
    if ccode = ccl then
      handle_file_error (old_discfile, 3);

    repeat

     { Read record from disk file, write employee name to $stdlist         }
     { and read corresponding record number from $stdin and update         }
     { the disk file with the input record and unlock disk file.           }

     read_length := FREAD (old_discfile, inbuf, 128);
     if ccode = ccl then
        handle_file_error (old_discfile, 4)
     else
        if ccode = ccg then
           end_of_file := true
        else
          begin
            FWRITE (std_list, inbuf, -20, octal('320'));
            if ccode <> cce then
              handle_file_error (std_list, 5);
            dummy := FREAD (std_in, inbuf[20], 5);
            if ccode = ccl then
              handle_file_error (std_in, 6)
            else
              if ccode = ccg then
                end_of_file := true;
            FUPDATE (old_discfile, inbuf, 128);
            if ccode <> cce then
              handle_file_error (old_discfile, 7);
          end
      until end_of_file;
    FUNLOCK (old_discfile);                 { final unlock of disk file    }
            if ccode <> cce then
              handle_file_error (file_num, 2);
  end;                                      { end update_file              }

  procedure close_disk_file
            (
               file_num : integer;
               disp     : integer
            );

  {**********************************************************************}
  {procedure close_disk_file is a generic file closing procedure that    }
  {allows you to specify the final disposition of the file you are closing. }
  {**********************************************************************}

  var
     msgbuf : pac80;

  begin
    FCLOSE (file_num, disp, 0);
    if ccode = ccl then
      handle_file_error (file_num, 8);
  end;                                     { end close_disk_file           }
```

```
{***************************************************************************}
{                              MAIN PROGRAM                                 }
{***************************************************************************}


begin
  filename := '&$stdlist&';
  open_file (std_list, filename, permanent,write,0,0);             { STEP 1}
  filename := '&$stdin&';
  open_file (std_in, filename, permanent,read,0,0);                { STEP 1}
  filename := '&dataone&';
  open_file (disk_file, filename, permanent,update,shared,locking);{STEP 1}
  update_file(disk_file);                                          { STEP 2}
  close_disk_file(disk_file, save);                                { STEP 3}

end.                                    { end main program              }
```

Index

A

aborting NOWAIT I/O, 19
access
  exclusive, 154
  modes, 179, 180
  multi, 153, 156, 157
  random, 119, 130, 131
  restricting, 179
  restricting by type, 181
  restricting by user, 181
  semi-exclusive, 155
  sequential, 119, 130
  shared, 156
  simultaneous, 153
access control definitions (ACD), 159, 178
accessing
  files, 18, 48, 51
  files, remote, 84
account security, 182
ACD
  adding, 166
  changes, logging, 177
  commands related to, 161, 162
  copying, 166
  creating, 162
  deleting, 166
  device owners, 160
  file owners, 160
  intrinsics related to, 161, 162
  listing, 164
  managing, 161, 162
  migrating, 167
  modes, 160
  modifying, 166
  operation, 160
  owners, 159
  pairs, 160
  preserving, 162
  replacing, 166
  scope, 159
  traditional security, and, 159
ACD (access control definitions), 159, 178
adding
  ACDs, 166
altering file use, 17
ASCII
  files, 61, 70
  transmission, 70

B

backreferencing files, 75, 76
binary
  files, 61, 70
  transmission, 70
BOT marker, 116
boundaries, half-word, 61

C

changing
  file domains, 89
  file security, 185
characteristics, files, 51
circular files, 60
class, volume, 63
closing
  files, 103, 106, 108
  permanent files, 106
  tape file, 107
  tape files, 107
command interpreter variables and
    expressions, 83
commands
  ACDs, and, 161, 162
comparing record types, 58
configuration, system, logging, 169
console
  requesting reply, 124
  writing messages to, 123, 124
copying
  ACDs, 166
creating
  ACDs, 162
  files, 17

D

data elements, 13
data transfer, 13, 109, 117
  intrinsics, 111
  multiple records, 113
default record size
  files, 62
  line printer files, 62
  magnetic tape files, 62
  plotter files, 62
  programmable controller files, 62
  synchronous single-line controller files, 62
  terminal files, 62
defining file characteristics, 18, 48
deleting
  ACDs, 166
  files, 89
device files, 15, 100

jobs and, 15
opening, 99
sessions and, 15
device owners
acds, 160
devices
ASCII transmission, 70
binary transmission, 70
EBCDIC transmission, 70
files, 15
peripheral, 13, 14
programs, 14
shareable, 18, 48
spooled, 15
directories, searching, 89
disk files, 15, 56
closing, 105, 107
opening, 94, 96
disk volume
specifying restrictions, 63
displaying
file equations, 193
file error information, 198
file information, 187
domains
changing, 89
new files, 87
permanent files, 88
temporary files, 87
duplicative files, 79, 197

E

EBCDIC transmission, 70
environment, remote, 84
EOT marker, 116
error check procedure, writing, 201
exclusive access, 154
expressions, 83
variables within file designators, 83

F

file
reading from labeled tape, 135
reading from tape, 134
file codes
reserved, 64, 69
specifying, 64, 69
file designators, 71, 85
file domains, 87, 90
changing, 89
features, 88
new files, 87
permanent files, 88
permitted, 88
temporary files, 87
file equations, 17, 75, 83
displaying, 193
file errors
displaying information, 198
file owners
acds, 160
file system interface, 13
files, 14
accessing, 18, 48, 51, 73, 75
accessing remote, 84
altering use, 17
ASCII, 61, 70
attributes, 17
backreferencing, 75, 76
binary, 61, 70
changing security, 185
characteristics, 51
circular, 60
closing, 103, 108
closing as permanent, 105
compatibility (pre- and post-900 series), 61
creating, 17, 74
default record sizes, 62
defining characteristics, 18, 48
deleting, 89
designators, 71, 83, 85
device-dependent characteristics, 99
devices, 15, 100
devices, opening, 99
disc, 56
disk, 15
displaying information, 187, 201
domains, 87, 90
duplicative, 79, 197
errors, displaying, 198
exclusive access, 154
getting information, 187, 201
half-word boundaries, 61
input, 14
interactive, 79, 197
jobs and sessions, 15
KSAM, 59
listing, 90
lockwords, 74, 75
magnetic tape, 56
mapped access, 141, 146
mapped, opening, 144
message, 60

Index

multi access, 153, 156, 157
multi access, global, 156
names, 14, 73
nonshareable, 15
opening, 91, 94, 102
output, 14
overrides, 55
parsing designators, 84
passed, 79, 81
passing, 79
peripheral devices, 14, 15, 56
permanent, 106, 188
predefined, 75
qualified, 74
random access, 121
reading from, 129, 136
record structure, 14, 18
records, 14
referencing, 73
reserved codes, 64, 69
rewinding, 115
RIO, 59
saving, 89
searchign directories, 89
security, 159, 184, 186
security, ACD, 159, 178
security, traditional, 179, 186
semi-exclusive access, 155
sequential access, 120
shared, 15, 153, 156, 157
sharing, hazards of, 157
simultaneous access, 153
specifying codes, 64, 69
specifying type, 59, 60
spooled (devices), 15
standard, 59
suspending and restoring security, 186
system hierarchy of overrides, 92
system-defined, 72, 77, 81, 97
tape, 100, 102, 107
temporary, 79, 193
types, 15
updating, 137, 139
user-defined, 72, 73, 76
validating designators, 84
writing to, 119, 128
FINFO function, 194
formats
comparison of, 58
fixed-length, 56
records, 56, 58
storage, 70
undefined-length, 56, 57, 58
variable-length, 56, 57
function FINFO, 194

G

getting file information, 187
global multi access, 156
group security, 183

H

half-word boundaries
files, 61
hazards of file sharing, 157
hierarchy
overrides, 55

I

I/O performance, increasing, 132
idx endNOWAIT I/O, 47
increasing I/O performance, 132
input
files, 14
NOWAIT I/O, 18
sets, 78, 79
standard, 77
interactive files, 79, 197
interface, file system, 13
intrinsics
ACDs, and, 161, 162
data transfer, 111

J

jobs
devices files and, 15

K

KSAM files, 59

L

labeled tape, 102, 126
reading from, 135
limitations
file designators, 73
file names, 73
NOWAIT I/O, 18, 19
line printer files
default record size, 62
listing
ACDs, 164
files, 90

lockwords
  changing, 75
  files, 74, 75
  removing, 75
log files
  specifying number of records, 69
logging
  ACD changes, 177
  file security events, 175
  password changes, 176
  printer access failure, 170
  process creation, 173
  restores, 169
  stream initiation, 172
  system events, 167
  system logging configuration, 169
  user, 174
  user logging, 173
long-mapped access, 144

M

magnetic tape, 115, 117
  default (file) record size, 62
  files, 56, 100
  marker, 115
  writing to, 125, 128
managing ACDs, 161, 162
mapped access
  advantages, 143
  long, 144
  opening files, 144
  restrictions, 142
  short, 143
  to files, 141, 146
marker
  BOT, 116
  EOT, 116
  magnetic tape, 115
message files, 60
messages
  writing to the console, 123, 124
migrating ACDs, 167
modes
  access, 180
  ACDs, 160
modifying
  ACDs, 166
moving a record pointer, 114
multi access, 153, 156, 157
  global, 156
  restrictions, 154
multiple records, data transfer, 113

N

names of files, 14
networks, NOWAIT I/O, 19
new files
  devices, 100
  domains, 87
  file domain, 87
nonshareable files, 15
NOWAIT I/O, 18
  aborting, 19
  input, 18
  intrinsics, 18
  limitations, 18, 19
  networks, 19
  output, 18

O

opening
  device files, 99
  files, 91, 102
  files, mapped, 144
  system-defined files, 97, 98
  tape files, 100, 102
operation
  ACDs, 160
output
  files, 14
  NOWAIT I/O, 18
  sets, 78, 79
  standard, 77
overrides
  file system hierarchy, 92
  files, 55
  hierarchy, 55
owners
  acds, 159

P

pairs
  ACDs, 160
parsing file designators, 84, 85
passed files, 79, 81
passwords
  changes, logging, 176
peripheral devices, 13, 56
  files, 14, 15
permanent files
  closing, 106
  devices, 100
  domains, 88
  file domain, 88

Index

plotter files
  default record size, 62
pointing to a record, 114
predefined files, 75
preserving ACDs, 162
printer
  access failure, logging, 170
procedure
  file system error check, 201
process
  creation, logging, 173
programmable controller files
  default record size, 62
programs
  devices, 14

Q

qualified file names, 74

R

random access, 119, 130, 131
  record selection, 111
reading from
  files, 129, 136
  tapes, 134
record pointers
  moving, 114
records
  comparison of, 58
  default sizes, 62
  files and, 14
  first, rewinding to, 115
  fixed-length, 56
  formats, 56
  pointers, 109, 110
  pointing to, 114
  selection, 110, 113
  selection, random access, 111
  selection, RIO access, 113
  selection, sequential access, 111
  selection, update access, 111
  spacing forward or backward, 114
  specifying size, 61, 62
  structure, 14, 18
  undefined-length, 56, 57, 58
  variable-length, 57
remote environment, 84
replacing
  ACDs, 166
requesting console reply, 124
reserved file codes, 64, 69
restore
  logging, 169
restoring security, 186
restricting access, 179
  by type, 181
  by user, 181
restrictions
  mapped access, 142
  multi access, 154
  sharing files, 154
rewinding files, 115
RIO
  access, record selection, 113
  files, 59

S

saving files, 89
scope
  ACDs, 159
searching directories, 89
security
  account level, 182
  changing, 185
  file level, 184
  files, 159, 186
  group level, 183
  logging file security events, 175
  restoring, 186
  suspending, 186
  traditional, 179, 186
semi-exclusive access, 155
sequential access, 119, 130
  files, 120
  record selection, 111
sessions
  device files and, 15
sets
  input, 78, 79
  output, 78, 79
  volume, 63
shared
  access, 156
  devices, 18, 48
  files, 15
sharing files, 153, 157
  hazards, 157
  restrictions, 154
short-mapped access, 143
specifying
  disk volume restrictions, 63
  file codes, 64, 69
  file types, 59, 60

record format, 56
record size, 61, 62
storage format, 70
specifying file domain, 87, 90
spooled devices, 15
spooling
console operator, 15
user, 15
standard
files, 59
input files, 77
output files, 77
storage formats, 70
specifying, 70
stream
initiation, logging, 172
suspending security, 186
synchronous single-line controller files
default record size, 62
system
events, logging, 167
system-defined files, 72, 77, 81, 97
designators, 77
opening, 97, 98

T
tape
labeled, 126
unlabeled, 125
tape files
closing, 107
magnetic, 100
opening, 100, 102
reading from, 134
temporary files
domains, 87
terminal files
default record size, 62
traditional file security, 179, 186
transmission
ASCII and binary, 70
EBCDIC, 70
types of files, 15

U
undefined-length formats, 56, 57, 58
unlabeled tape, 125
update access
record selection, 111
updating files, 137, 139
user
logging, 174
logging, logging, 173
spooling, 15
user-defined files, 72, 73, 76

V
validating file designators, 84, 85
variable-length formats, 56, 57
variables, 83
expressions within file designators, 83
volume
class, 63
set, 63
single, 63

W
writing file system error check procedure, 201
writing to
files, 119, 128
magnetic tape, 125, 128