HP 3000 Computer Systems

File System: Reference Manual

HEWLETT
PACKARD

HP Part No. 30000-90236
Printed in U.S.A. 1989

Third Edition
E1089

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 AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein 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.

Print History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	**Date**	**Software Version**
First Edition	February 1982	
Second Edition	June 1987	G.03.00
Update 1	October 1988	G.03.04
Third Edition	October 1989	G.03.08

MPE V Manual Plan

INTRODUCTORY LEVEL:

GENERAL INFORMATION
Manual
(5953-7553)

MPE V/E HP 3000:
Fundamental Skills
(Part No. Pending)

HP 3000 GUIDE
FOR THE
NEW SYSTEM OPERATOR
(32033-90021)

MPE V/E HP 3000:
Advanced Skills
(Part No. Pending)

MPE V GENERAL USER'S
Reference Manual
(32033-90158)

FCOPY
Reference Manual
(32212-90003)

STANDARD USER LEVEL:

MPE V COMMANDS
Reference Manual
(32033-90006)

MPE V INTRINSICS
Reference Manual
(32033-90007)

MPE V SYSTEM UTILITIES
Reference Manual
(32033-90008)

MPE V SEGMENTER
Reference Manual
(30000-90011)

MPE V DEBUG/
STACK DUMP
Reference Manual
(30000-90012)

MPE V FILE SYSTEM
Reference Manual
(30000-90236)

ADMINISTRATIVE LEVEL:

MPE V SYSTEM
OPERATION AND
RESOURCE MANAGEMENT
Reference Manual
(32033-90005)

MPE V SECURITY AND
ACCOUNT STRUCTURE
(32033-90136)

MPE V STORING AND
RESTORING FILES
(32033-90133)

MPE V SYSTEM BACKUP
AND RECOVERY
(32033-90134)

SUMMARY LEVEL:

MPE V QUICK
REFERENCE GUIDE
(32033-90023)

LG200027_009a

Preface

The File System Reference Manual is the reference for MPE (Multi Programming Executive) V/E operating system on the HP 3000.

You are assumed to have a working knowledge of the language(s) to be used and of the MPE V/E operating system.

To assist you in locating information, a brief description of each chapter in this manual follows:

Chapter 1 Introduction provides an overview of the *MPE File System Reference Manual.*

Chapter 2 Record Structure describes the data fields which are organized into logical records and discusses the best approach to efficient blocking.

Chapter 3 File Structure discusses the file size, the arrangements of files in extents, file identification and specification of file characteristics.

Chapter 4 Domains describes the classifications of files as new, temporary or permanent.

Chapter 5 File Operation describes the usage operation of files.

Chapter 6 Data Transfer describes the selection of records and the transfer of information, including the use of buffers and considerations for shared files.

Chapter 7 File Security discusses the security provisions and specifies any restrictions on access to the files associated with each account, group and individual file.

Chapter 8 Interprocess Communications describes the interprocess communications (IPC) facility of the file system which permits multiple process to communicate with one another.

Chapter 9 Magnetic Tape Considerations discusses the magnetic tape storage medium and the requirements associated with magnetic tape files.

Appendix A File System Reference provides a summary of the manual's information.

Appendix B Status Information describes how to check disk file status to include their physical characteristics, current file information and error information.

Appendix C Terminal Characteristics provides reference material for terminals and character printers.

Appendix D ASCII Character Set provides the available ASCII character set used with MPE V/E.

Appendix E Disk File Labels describes the contents of the disk file labels.

Appendix F End-of-File discusses the use of the end-of-file indicator.

Appendix G Magnetic Tape Labels describes the contents of the magnetic tape labels.

Some additional sources of information that might be helpful include:

- *MPE V System Operation and Resource Management Reference Manual* (32033-90005)
- *MPE V Commands Manual* (32033-90006)
- *MPE V Utilities Reference Manual* (32033-90008)
- *MPE V Intrinsics Reference Manual* (32033-90007)

Conventions

UPPERCASE

In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

```
COMMAND
```

can be entered as any of the following:

```
command         Command         COMMAND
```

It cannot, however, be entered as:

```
comm            com_mand        comamnd
```

italics

In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

COMMAND *filename*

bold italics

In a syntax statement, a word in bold italics represents a parameter that you must replace with the actual value. In the following example, you must replace ***filename*** with the name of the file:

COMMAND(***filename***)

punctuation

In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

(*filename*):(*filename*)

underlining

Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

```
Do you want to continue? >>  yes
```

{ }

In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either ON or OFF:

$$\texttt{COMMAND}\begin{Bmatrix}\texttt{ON}\\ \texttt{OFF}\end{Bmatrix}$$

[ ]

In a syntax statement, brackets enclose optional elements. In the following example, OPTION can be omitted:

COMMAND *filename* [OPTION]

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select OPTION or *parameter* or neither. The elements cannot be repeated.

$$\texttt{COMMAND}\ \textit{filename}\ \begin{bmatrix}\texttt{OPTION}\\ \textit{parameter}\end{bmatrix}$$

[...]

In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

[,*parameter*] [...]

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter:*

[*parameter*] [, ...]

| ... |

In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select A, AB, BA, or B. The elements cannot be repeated.

$$\begin{Bmatrix}\texttt{A}\\ \texttt{B}\end{Bmatrix}\ |\ \ldots\ |$$

...

In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

Δ

In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

(*parameter*)Δ(*parameter*)

▭

The symbol ▭ indicates a key on the keyboard. For example, RETURN represents the carriage return key or Shift represents the shift key.

CTRL *character*

CTRL *character* indicates a control character. For example, CTRL Y means that you press the control key and the Y key simultaneously.

Conventions (continued)

base prefixes

The prefixes %, #, and $ specify the numerical base of the value that follows:

%num specifies an octal number.
#num specifies a decimal number.
$num specifies a hexadecimal number.

If no base is specified, decimal is assumed.

bits (*bit:length*)

When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format bits (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, bits (13:3) indicates bits 13, 14, and 15:

```
most significant                least significant
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0|  |  |  |  |  |  |  |  |  |  |  |  |13|14|15|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
bits (0:1)                             bits (13:3)
```

Contents

Figures

Tables

Section Divider

1. Introduction

1

Introduction

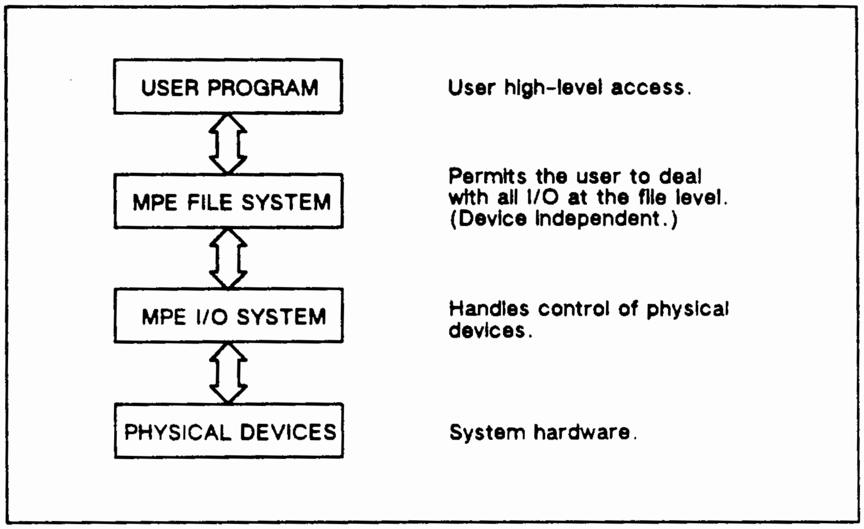
This manual describes the MPE file system. The file system is the part of the MPE operating system that manages information being transferred or stored with peripheral devices. It handles various input and output operations, such as the passing of information to and from user processes, compilers, and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

Since all input and output operations are performed through the use of files, you may access very different devices in a standard, consistent way: it will not make much difference to you whether you read your file from a disk, from a magnetic tape, or from cards, 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 will determine 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, not even the MPE file system will permit you to read a file from a line printer.) You can use the MPE `FILE` command to specify the device you want.

Figure 1-1 shows how your program, the file system, the I/O system, and the actual hardware of the system are related.

Notice that the file system serves as the interface between you and the rest of the system.


USER PROGRAM
User high-level access.
MPE FILE SYSTEM
Permits the user to deal with all I/O at the file level. (Device independent.)
MPE I/O SYSTEM
Handles control of physical devices.
PHYSICAL DEVICES
System hardware.

LG200016_001

Figure 1-1. File System Interface

Section Divider

2. Record Structure and Blocking

2

Record Structure and Blocking

This chapter will discuss record structures, record lengths, blocking factors and system consideration of the various formats used.

Logically related data elements are grouped together, forming a logical record, the fundamental unit of information that is handled by the MPE File System. It is the smallest data grouping that can be defined by the user to the File System.

Since a logical record is a group of various data elements, its structure will depend on the number, content and size of the data elements within it. Therefore, when you design your records, there are several questions to consider:

- How will the data be represented?
- Will all the records be the same size?
- How long will the records be?
- Should logical records be grouped together for transfer?

In this chapter we will discuss these questions.

Data Representation: ASCII Versus Binary

Devices on the HP 3000 can transmit information in ASCII (American Standard Code for lnformation Interchange) in binary code, or both formats depending on the device. For example, a line printer handles ASCII formatted data, while a disk can transmit and store data in either format.

Note

It is possible to transmit and store data in EBCDIC, as long as the application program or subsystem (FCOPY, for example) handles the character translation. EBCDIC is not handled automatically by MPE.

With many devices, there is no format 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 you want, or accept the MPE default for the device you are using.

The distinctions made between ASCII and binary files do not affect the record size determination.

When the allocated record space is not filled by data, records are padded with blanks for ASCII files and zeros for binary files; this padding is the only significant difference between ASCII and binary files.

Examples of ASCII files on the HP 3000 include program source files, general text and document files, and MPE stream files containing MPE commands. Examples of binary files include USL (User Subprogram Library) files containing compiled object code, program files containing prepared object code, and application data files. In MPE, printers, plotters, and card readers are accessed as files.

Record Formats

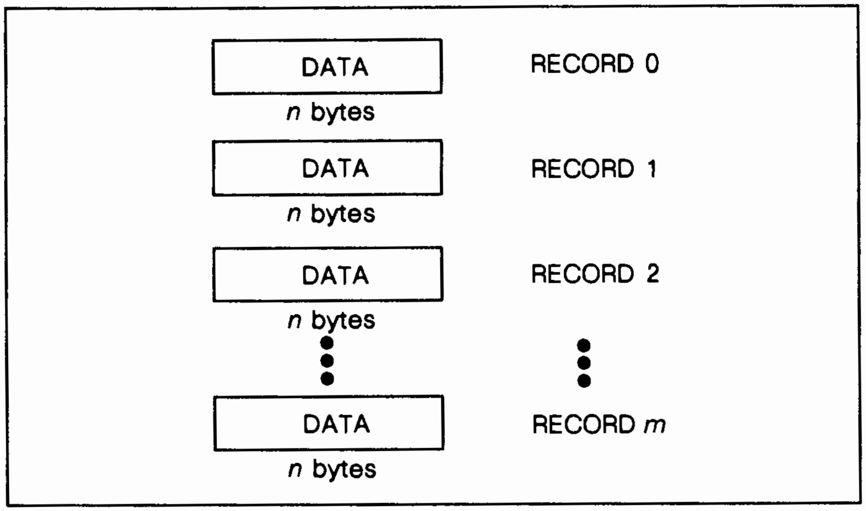
A file can contain records written in only one of three formats: fixed-length, variable-length and undefined-length. You can specify the format you want for your records, either with the FOPEN intrinsic or with the MPE `BUILD` or `FILE` command. `FOPEN`,

Files residing on disk or magnetic tape may contain records in any of the three formats. For files on other devices, the file system will override any specifications you supply, and will treat 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 will know how much space has been allocated for each record, and that all of the space is to be available for data.

Figure 2-1 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.



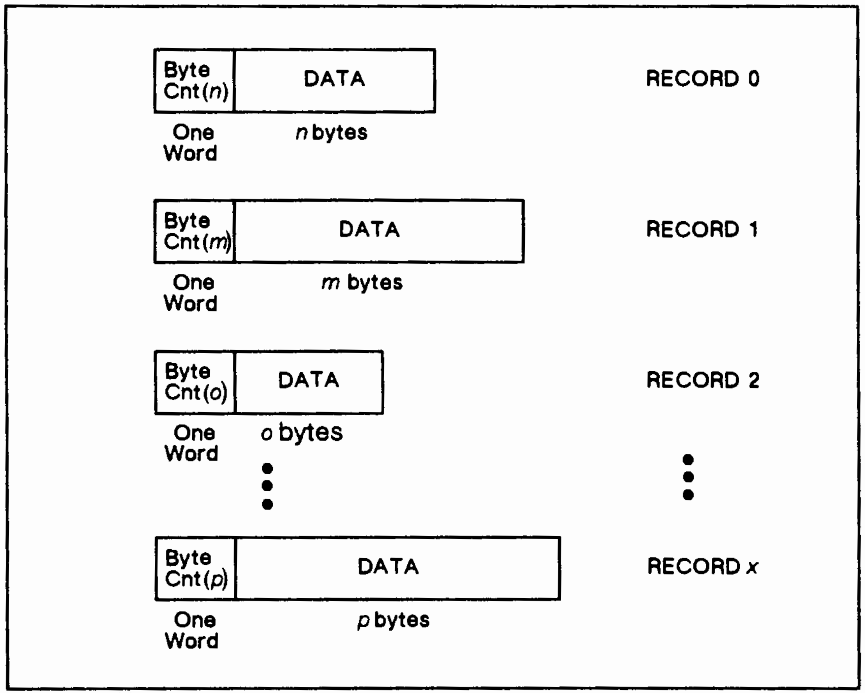
LG200016_002

Figure 2-1. Fixed-length Records

Variable-Length Records

If you want a disk file in which the logical records need not be the same size. You can request that the format of the records be variable-length. Each record is preceded by a one-word (16-bit) counter giving the length of the record in bytes. In variable-length format 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 2-2 shows a file with variable-length records. A one-word byte precedes the first word of each record indicating the length of the record.



LG200016_003

Figure 2-2. Variable-length Records

Undefined-Length Records

Undefined records are useful for terminals and magnetic tapes. You can write various size records to tape. When a record is read back, its size is made determined by the hardware and made available to the program. The size of records read from a terminal is the number of characters preceding a carriage return.

The three record formats, fixed-length, variable-length, and undefined-length, are summarized in Table 2-1.

Table 2-1. 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.	Record length varies.
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.

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 `FOPEN` intrinsic. The file system uses the convention that a negative record size means the size is given in bytes and a positive record size means it is in words. The record size may be given in either words or bytes, regardless of whether the file is to be represented in ASCII or binary code. 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 and in various subsystems, the record size for an ASCII file is usually identified in terms of bytes and the record size for a binary file is identified in terms of words. This convention is a matter of convenience only, since most users think of ASCII files as being character oriented.

The HP 3000 is designed around a 16-bit word boundary. Records are aligned on the predetermined boundaries. This has a particularly important effect on disk and magnetic tape files. Odd byte record lengths are always grouped so that logical records begin on word boundaries. When the file is a binary file, the extra byte is available to be used for data. Similarly, for variable-length ASCII files, the odd byte length are grouped and accessible for data.

However, if the file is ASCII and has fixed-length or undefined-length records, the extra byte is not accessible for data. The odd byte length remains the maximum size allowed for data. Figure 2-3 shows the placement of odd byte records and the disposition of the added byte.

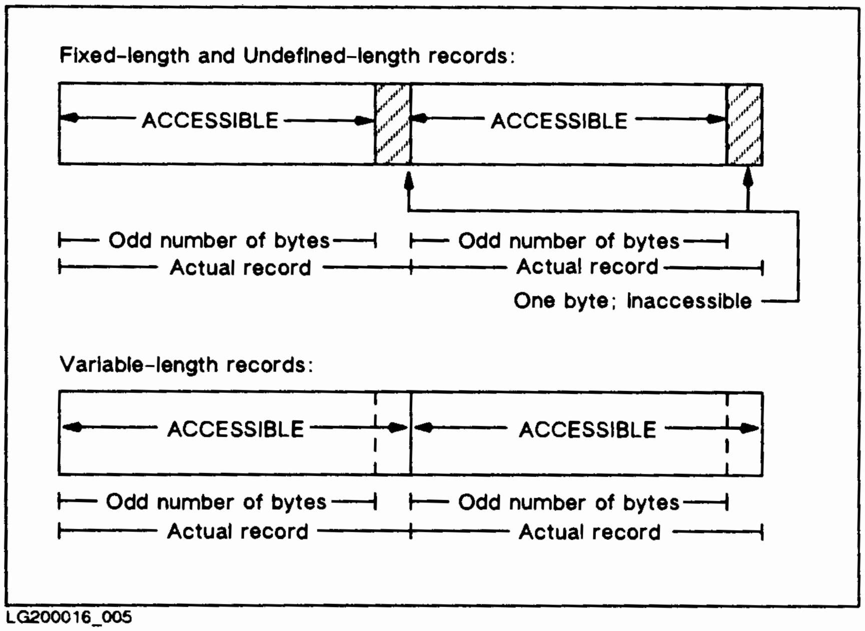


Figure 2-3. Record Placement for ASCII Files

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

Table 2-2. Standard Default Record Sizes

DEVICE	RECORD SIZE (BYTES)
Disk	256
Magnetic Tape Unit	256
Terminals (most cases)	80
Card Reader	80
Line Printer	132
Paper Tape Reader	80
Paper Tape Punch	256
Plotter	510
Printing Reader/Punch	No. of card columns, usually 80
Programmable Controller	256
Synchronous Single-Line Controller	256

Physical Records and Blocking

The logical record is the smallest data grouping you may directly address. The physical record, or block, is a grouping of one or more logical records, and is the unit of information moved in one physical read or write of data to or from the device containing the file.

The file system automatically handles the blocking and deblocking of logical records when you are operating in buffered mode, but you can block the records in your files on disk or tape when you are operating in NOBUF mode. For files on other devices, physical records are blocked according to the characteristics of the devices. A physical record may be one card, or one line of print.

You may specify the blocking factor, or number of logical records in a block, for the records in your files with the FOPEN intrinsic or the BUILD or FILE command. The maximum blocking factor is 255. The actual structure of your blocks will depend upon the format of your logical records; for example, a block of fixed-length records will be structured differently from a block of variable-length records.

Efficient grouping of logical records into blocks results in:

- Fewer disk accesses.
- Better disk space utilization.

Note

Once the blocking factor is set, it cannot be overridden during the life of the file.

Disk Access Considerations

Since one disk read or write will transfer one physical record, you can minimize input and output and file system overhead by grouping several logical records into each block: one read or write will transfer as many records as you have in one block.

In MPE, blocking helps to minimize device head contention on system domain disks and shared private volumes. Since one disk access will read or write as many records as one block contains, you and other system users will need to gain control of the disk less often than if you had to read each logical record individually.

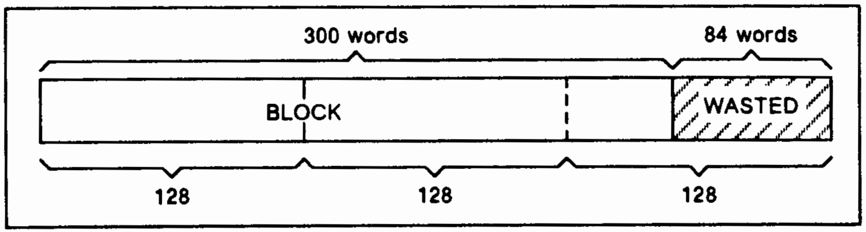
Note

The block size of your file is determined by multiplying two parameters you supply when you create the file: the record size and the blocking factor. The maximum block size allowed is 32,000 words.

Disk Space Considerations

You can save or waste disk space in the way your files are blocked. Disks are not word addressable, disk space is organized into physical groups of 128 words called sectors. Because of this organization, all physical transfers must begin on a sector boundary, and so all physical records (blocks) must begin on a sector boundary. A physical record may span more than one sector, but it must begin on a sector boundary. If your blocks do not fit neatly into sectors—that is, if their size is not a multiple of 128 words—some disk space will be wasted.

Suppose your blocks are 300 words long. They will each cover two sectors and part of a third, as shown:



LG200016_006

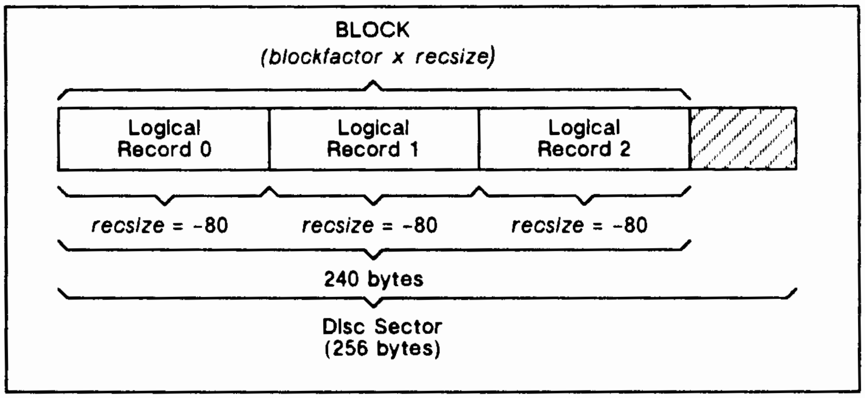
Since each block starts on a sector boundary, the unused space following each block is wasted. If the block size had been 384 words rather than 300, all of the space in three sectors could be used for each block, and no space would be wasted.

It is recommended that to minimize wasting file space, choose a blocking factor that is equal to or slightly less than a multiple of 128 words.

Blocks Containing Fixed-Length Records

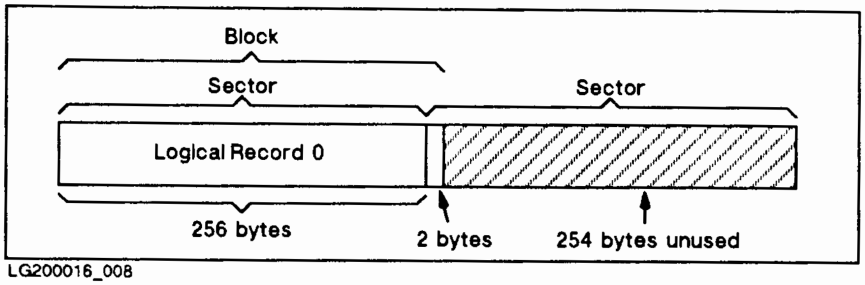
When your file contains fixed-length records, all the blocks will contain the same amount of data in the same number of records.

The file system determines the size of your blocks by multiplying two parameters you supply: the logical record size (recsize) times the blocking factor. When written to disk, each block begins at the start of a sector and may occupy one or more contiguous sectors. This occurs because the disk controller can only transfer data in units equivalent to the length of a sector, 128 words or 256-bytes. Thus, on a disk file, a 240-byte block containing three 80-byte fixed-length records would appear as follows:



LG200016_007

The 16 bytes remaining at the end of the sector are wasted, since this block cannot use them and the next block begins at the start of the next sector. Because of this fact, you can waste disk space if you do not block your records carefully. For example, if you use a blocking factor of 1 when writing a fixed-length record of 258-bytes, you will waste 254 bytes of disk space as shown below:



LG200016_008

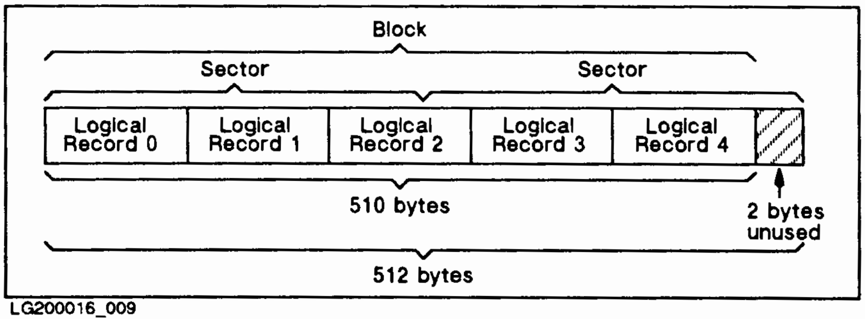
In a large file, this much waste (about 49.6%) soon becomes devastating. For optimum use of disk space, compute the block size so that:

recsize x *blockfactor* = a multiple of 256 (for bytes)

or:

recsize x *blockfactor* = a multiple of 128 (for words)

If you can't make the blocks fit into sectors exactly, it is better to have blocks a bit too small than too large; less space is wasted. For example, if your records are 102 bytes long, there will be little waste if you choose a blocking factor of 5: 102 x 5 = 510, so your block will occupy two sectors with only two wasted bytes, as shown below:



LG200016_009

Blocks Containing Variable-Length Records

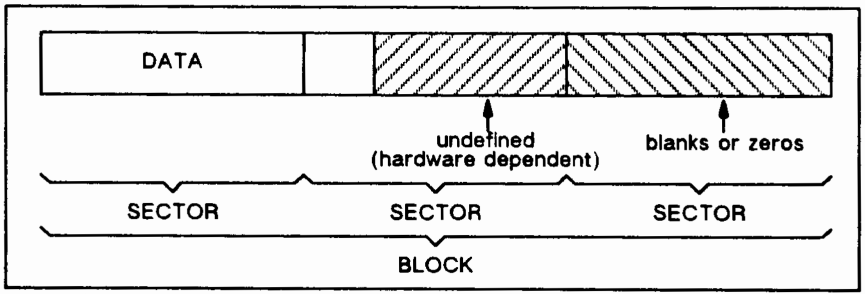
When your file contains variable-length records, one block may contain a variable number of records: one block may contain a few large logical records or many small ones. The same amount of space will be available for each block, but since the logical records will be of different sizes.

The file system will change the record size and blocking factor that you specify for your blocks. Your record size and blocking factor will be multiplied to yield a new record size, and your blocking factor will be changed to 1. For example, if you request a record size of 20 words and a blocking factor of 6, your new record size will be 120 words and your blocking factor will be 1. In both cases, the available space in your blocks is the same: 120 words. The record size and blocking factor are manipulated this way simply to maintain a consistent internal structure. The actual block size will be several words larger than the available space: the file system adds one word for a byte count at the beginning of each record and another word for a delimiter of "−1" at the end of each block. These words of overhead allow for a minimum of one logical record per block.

Your block size will be determined by the formula:

(*recsize* + 1) x (*blockfactor* + 1)

To avoid permanent waste of file space, make your block size equal to or slightly less than, a multiple of 128 words. This will make most of the space covered by your file available for data. Even if your blocks fit neatly into sectors, however, disk space may be wasted if your logical records fit into your blocks poorly. Here, two logical variable-length records fit into a block. A third record is too large to fit into the block, so space is wasted:

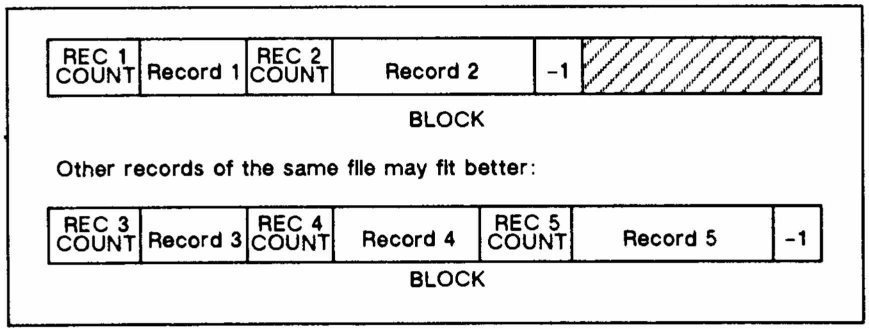


LG200016_011

Blocks Containing Undefined-Length Records

When your file contains undefined-length records, it is impossible to take advantage of blocking. Since the file system does not know how much space the actual data will require, it cannot place more than one record in a block; it does not know that more than one record will fit into each block. For this reason, the file system will override any blocking factor you supply and change the blocking factor to 1. Each block will contain one record. When you create your file, specify a logical record size large enough to contain the largest record you expect; the file system will allocate this much space for each block. In files containing undefined-length records, logical records and physical records are identical.

To avoid permanent waste of sector space in your file, your block size (record size) should be a multiple of 128 words (256-bytes). Records that are considerably shorter than the size allowed will waste space, since the maximum record space is allocated and may contain only one record. Consider the case below:



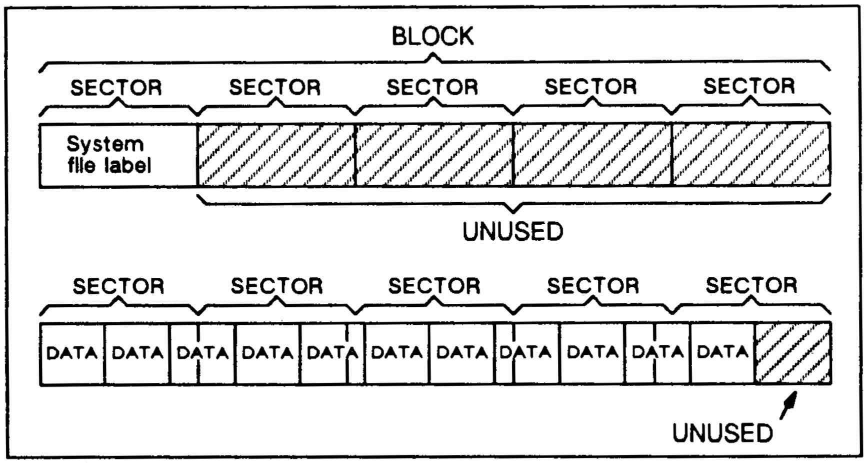
LG200016_010

A block size of 384 words (768-bytes) has been specified; each block covers three sectors. The record in this case is only 140 words long. It fills one sector, and ends in the next; the contents of the remainder of the second sector is hardware dependent. The third sector is not used at all, and is filled with blanks (if this is an ASCII file) or zeros, if the file is written in binary code.

Blocking Consideration: System File Label

Every disk file has a system file label. The label identifies the file to the system and contains the characteristics of the file. The system file label is 128 words long, and occupies one block. An entire block is allocated for the system file label for the sake of uniformity. All blocks in the file are treated the same way. Because of this fact, a small amount of file space can be wasted even if your records fit neatly into blocks. Since the system file label requires only 128 words, any space beyond that in its block may be wasted.

Consider the extreme case of a file with 11 records, each 55 words long. If a blocking factor of 11 is chosen, one block will contain the 11 records. The block will cover five sectors for a total of 640 words, so only 35 words will seem to be unused. However, a block of the same size is allocated for the system file label. Since it requires only 128 words, the additional 512 words in the block are unused. The waste in this case includes the 35 words left over by the records plus the 512 words lost on the system file label. A total of 547 words wasted. This rather extreme situation is illustrated below:



LG200016_012

Relative I/O Block Format

Relative I/O format is a scheme (used principally by COBOL) for tagging each record with a bit describing whether a record is active. Records can be logically deleted from the file by setting their activity bits to inactive status. The blocks used with relative I/O have a characteristic format. This format is illustrated in Appendix A, "File System Reference".

Improving Input/Output Efficiency

When you run a program that transfers data to or from a different input or output device from time to time, you can make the physical input or output more efficient by overriding the programmatically-specified record size or blocking factor so that these values better suit the device involved. For instance, suppose you are running a program originally written to read input from cards specified as 20-character logical records. If, before the next run, the input file has been copied to disk, you could provide faster access by reading these records in blocks of 240 characters. To do this, you would enter a FILE command using a blocking factor of 12:

```
FILE CARDS; DEV=DISC;REC=-20,12
RUN PROGX
```

Note

When you specify record size in bytes, you must precede the block factor value with a minus sign when you express record size in words, be sure to omit the minus sign.

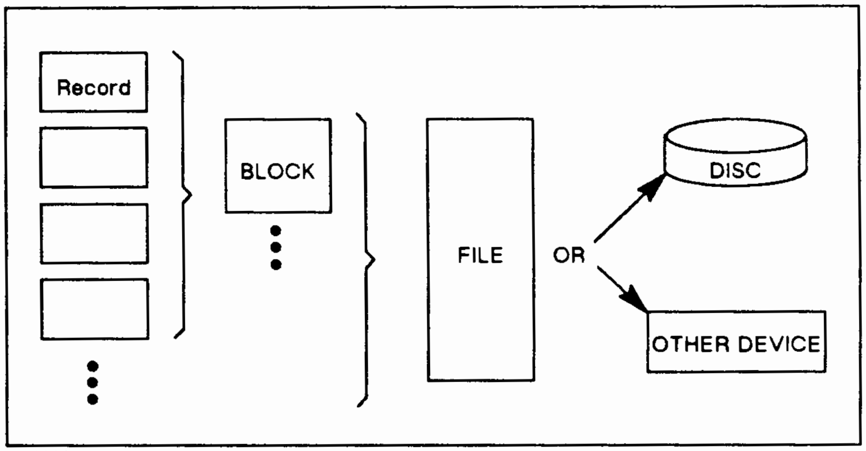
Section Divider

3. File Structure

3

File Structure

In this chapter, we will investigate files, sets of logically related records which reside on one or more devices. Records are related to files as illustrated in Figure 3-1, below.



LG200016_013

Figure 3-1. Records/Files Relationship

When you design your files, there are several questions you should consider:

- Where should the file be kept?
- How large should the file be?
- How should a disk file be distributed on the disk?
- How should the file be identified?
- What characteristics should the file have?

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:

- Disk files , which are files residing on disk, immediately accessible by the system and potentially shareable by several sessions/jobs at the same time.
- Device files , which are files currently being input to or output from any peripheral device except a disk. (Files on serial disk are considered device files.) When information exists on such a device but is not being processed, the File System cannot recognize it as a file. Thus, information on cards is not identified as a file until the cards are loaded into the card reader 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 accessed exclusively by the session or job that acquires it and is owned by that session/job until the session/job 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 non-disk devices. The console operator, not the user, controls the spooling operation.

File Placement

Free space on a disk is often not contiguous. It exists as small chunks of space between occupied spaces. Therefore, when you create a file and request a certain amount of space for it, the file system breaks your file into individually placeable pieces called extents. These extents may be placed wherever they can fit on the disk, or may even be scattered over several disks, and the file system will recognize them as belonging to the same file. Space for each extent will be allocated and initialized as it is required.

Note

When you create your file, make it as large or larger than you believe it will ever need to be; later, you can make a file smaller, but you cannot request more space for it unless you purge and recreate it. If you use only a part of the file space you have requested, you can access the file later and append to it, but you may only fill the file to the limit you set when you create the file.

Extents

Each extent is an integral number of blocks; it consists of a number of consecutively located disk sectors. You may specify the maximum number of extents your file may occupy, up to a maximum of 32. The file system, however, may use fewer extents than you request. Each extent must contain at least one physical record. So, if your file consists of one block of data and one block for the system file label, and you request eight extents for your file, the file system will allot only two extents.

Each extent is the same size, with the possible exception of the last. If the records cannot be distributed evenly among the extents, the last extent will receive fewer records than the others. You can determine the size of each extent by the following method.

In this algorithm, the constant 256 denotes the size of each sector in bytes:

If any division equates

Extent Size = Sectors/Extent

```
Sectors/Extent = Number of blocks
   divided by Number of extents
   multiplied by Sectors/Block

Sectors/Block = Block size (in bytes)
  divided by 256

Number of blocks = Number of records
     divided by Block factor
     plus 1 (for file label)
```

If any division equates in a fractional remainder, the quotient is rounded to the next higher integer (e.g., 65/8 = 8 with a fraction of 1 remaining, this equation is rounded up to 9).

To illustrate the use of the algorithm, the extent size is calculated below for a file containing 1024 logical records, organized as eight extents, with a blocking factor of 3. Each record is an 80-byte card image. The extent size is:

```
Number of blocks = 1024 divided by 3 + 1 = 343

Sectors/Block = 240 divided by 256 = 1

Sectors/Extent = 343 divided by 8 x 1 = 43 x 1 = 43

Extent Size = 43 Sectors/Extent
```

The first seven extents contain 43 sectors each and the last extent contains 41 sectors.

With a blocking factor of 16, applied in the above example, the extent size is:

```
Number of blocks = 1024 divided by 16 + 1 = 65

Sectors/Block = 1280 divided by 256 = 5

Sectors/Extent = 65 divided by 8 x 5 = 9 x 5 = 45

Extent Size = 45 Sectors/Extent
```

In this case, the first seven extents contain 45 sectors each, and the last extent contains 10 sectors.

Note

Spooled device files (spoolfiles) are written in a special format and managed entirely by the file system. They, like other files, may have a maximum of 32 extents. The size of these extents is determined by the system manager when configuring the system.

The extents that comprise your file will reside on disks in the device class that you specify when you open the file. Normally, each extent is assigned arbitrarily to a device in the class. If you wish, you may have all your extents on the same disk by requesting a specific logical device by its logical device number (ldev#) rather than by device class. In either case, MPE maintains the integrity of your extents. Any extent resides entirely on one device, and is not shared over several.

If your system contains two or more disks with the class name DISC, and their speeds are different, you may wish to consider the way your file will be used and choose one disk specifically because of its speed. In this case, you would reference that disk by its logical device number (ldev#) when you create the file.

Device class names and logical device numbers are discussed later in this chapter.

Extent Allocation

When you create your file and specify the number of extents you want, you may not need all of those extents right away. Perhaps you will need only a small part of the file space at first, and do not anticipate filling the space until later. In this case, when you build your file you can specify how many extents are to be allocated at once; the file system default is one extent. As the file grows to its full size and requires more space, the file system allocates the extents you have reserved as they are needed. This ability to take only as much space as you need when you need it enables you to optimize file access to save disk space.

When you create your file, only the space that is actually allocated is subtracted from the total available to you. The file system will allocate only as many extents as you request, but will remember how many extents you will eventually want.

If you create a file that will eventually consume more disk space than you have in your group or account, extents will be allocated until your available space is filled. After that, you may not allocate more extents until you increase the disk space available to you.

Occasionally, you may wish to override programmatic specifications dealing with extents to optimize the use of disk space. For instance, if your disk space is limited, the space available may exist as isolated small groups of sectors (fragments) rather than as contiguous groups of many sectors. You may then decide to break the file into more extents, each small enough to fit into the fragments available. If you fail to do this, perhaps attempting to open the file with one extent, you may not be able to get the disk space you require. To illustrate, consider that your disk space is limited and you run a program to create a new file for 1000 records of 80 bytes each, treated as 10 extents, you could enhance yhour chances of acquiring the disc space needed by issuing a :FILE command specifying 32 extents, all allocated immediately:

```
 No. of extents allocated immediately
            No. of extents in file  |
                File capacity   |   |
                            |   |   |
                            ▼   ▼   ▼
:FILE MYFILE;DEV=DISC;DISC=1000,32,32
:RUN XPROG       ▲
                 |
                 Requests search for space on any disk
```

In general, the rule is that if your disk space is limited and you know the total space your file will need, divide the file into many extents and request immediate allocation of all of them.

When you create your file and allocate the initial extents, those extents are allocated in order. Subsequent allocations of extents for that file need not be in order. For example, if you write a record that maps into an unallocated extent, that extent will be allocated, but intervening extents will not.

Extents are allocated as they are needed until your file has grown to its full size. When your file's initially allocated extents have been filled, the next block you write to the file forces allocation of an extent for that block. Similarly, if you try to read from an unallocated block, the extent containing that block will be initialized and allocated.

The file system will not allow uninitialized space to be read. When an additional extent is allocated for a file, it is initialized before it can be read; this is done for security reasons, to prevent a user from reading information that he did not write, such as old (purged_ files. The file system assumes that any space which is beyond the end-of-file indicator or which has not yet been allocated is uninitialized.

Note

Since extents need not be allocated in order, the last extent of a file may be allocated before the middle extents. For this reason, even if the end-of-file indicator is set at the file limit, the file system will not assume that the entire file space has been allocated and initialized.

Performance Implications of Extent Allocation

You can take advantage of the ability of the file system to allocate and initialize extents as they are needed. Instead of initializing your extents when you open your file, distribute the file system overhead (that is, the wait time) by initializing the minimum amount of space needed. When you write data to your file, you do not need to initialize the file space; this is only done when you attempt to write a new extent. So, by initializing only a minimum of space, you avoid the unnecessary initialization of file space that will be written to, and save time.

Special Considerations for Program Files

Program files must be contained in only one extent. When the MPE Segmenter prepares a program file, the file is automatically created with one extent, and so it fits this specification. If you are preparing a program onto an existing file in your job/session domain, you can make sure the file is limited to one extent by using the BUILD command:

```
:BUILD PROGFL;DISC=,1; CODE=PROG
                    ↑ Specifies 1 extent
```

Defining File Characteristics

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 characteristics are determined by the parameters you choose when you create the file with the FOPEN intrinsic or BUILD command, or when you specify the file with the FILE command. Once a file has been created, its characteristics cannot be changed. It can be renamed, purged, or made permanent, but the only way to change the attribute is by building a new file and copying the old one into it.

FOPEN Intrinsic

The FOPEN intrinsic is your best tool for supplying the file system with information about your file.

The FOPEN intrinsic is used to define the structure of the file and its records, and the file's identification, domain and usage. The characteristics that are affected are listed in Table 3-1, along with the corresponding FOPEN parameters and their defaults.

Table 3-1. FOPEN Parameters and Their Defaults

Record Structure	FOPTIONS RECSIZE	Binary, Fixed Recsize = 128 words
File Structure	BLOCKFACTOR FILESIZE NUMEXTENTS INITALLOC DEVICE FOPTIONS	(128/RECSIZE), rounded down 1023 records 8 extents 1 extent Disk

File Identification	USERLABELS
	FILECODE
	FORMALDESIGNATOR
	FOPTIONS
		Userlabels= 0
		Filecode= 0
		Unnamed

File Domain	FOPTIONS	New
File Usage	AOPTIONS NUMBUFFERS FOPTIONS	Read Only Access= SHR (read only) EXC (all others) No FLOCK Buffered No mulitrec No multi-access Wait for I/O Numbuf=2

File domains are discussed in a later chapter. Settings for the aoptions and foptions are shown in Appendix A. For more details on using the FOPEN intrinsic, see the *MPE V Intrinsics Reference Manual* (32033-90007).

BUILD Command

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

The parameters for the BUILD command have meanings and applications that are similar to the corresponding parameters for FOPEN. For more information about how to use the BUILD command, see the *MPE V Commands Reference Manual* (32033-90006).

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 FOPEN or BUILD, but you cannot actually create a file with the FILE command. While FOPEN and physically allocate space for a file and define its characteristics. The FILE command will define file attributes that will be ascribed to when a program accessing it is run. A comparison of the parameters for FILE and FOPEN is given in Table 3-2.

Table 3-2. FILE and FOPEN Parameters

Characteristic	FILE parameter	FOPEN parameter	MPE default
Formal file designator	*formaldesignator*	*formaldesignator*	Temporary nameless file.
Actual file designator	*filereference* $NEWPASS $OLDPASS $NULL $STDIN $STDINX $STDLIST	Default file designator *foption* (bits 10:3)	Same as formal file designator
Domain	NEW OLD OLDTEMP	Domain *foption* (bit 14:2)	New file
Logical record size	*recsize*	*recsize*	Configured default size of device for unit-record devices; 256 bytes for other devices
Block/buffer size	*blockfactor*	*blockfactor*	Configured block size of device divided by recsize
Record format	F V U	Record format *foption* (bit 8:2)	Fixed-length records for disk and magnetic tape files; undefined-length records for all others
ASCII/Binary Code	ASCII BINARY	ASCII/Binary *foption* bits 13:1)	Binary
Carriage-control characters supplied in FWRITE	CCTL NOCCTL	Carriage-control *foption* (bits 7:1)	No carriage control characters supplied in FWRITE.
Access mode	IN OUT OUTKEEP APPEND INOUT UPDATE	Access-type *aoption* (bit 12:4)	Read-only access for all devices except output devices that are assigned output-only access
Number of Buffers	*numbuffers* NOBUF	*numbuffers* (bits 11:5)	2 buffers

Table 3-2. FILE and FOPEN Parameters (continued)

Characteristic	FILE parameter	FOPEN parameter	MPE default
Exclusive/Share access	EXC SEMI SHR	Exclusive access *aoption* (bits 8:2)	For read-only access, SHR takes effect; for other modes, EXC
Multi-access	MULTI NOMULTI GMULTI	Multi-access mode *aoption* (bits 5:2)	No multi-access allowed
Multi-record mode	MR NOMR	Multi-record *aoption* (bits 11:1)	No multi-record mode
File disposition	DEL SAVE TEMP	(None-defined by default disposition parameter of FCLOSE)	Same as when file was opened
Device Class Name or Logical Device Number	*device*	*device*	Class Name DISC
Output priority	*outputpriority*	*numbuffers* (bits 0:4)	8
NOWAIT input/output	NOWAIT WAIT	NOWAIT I/O *aoption* (bits 4:1)	NOWAIT input/output prohibited
Number of copies	*numcopies*	*numbuffers* (bits 4:7)	1
File code	*filecode*	*filecode*	0
File capacity	*numrec*	*filesize*	1023
Total number of extents	*numextents*	*numextents*	8
Extents initially allocated	*initalloc*	*initalloc*	1
FILE command prohibition	(None)	Disallow FILE equation *foption* (bits 5:1)	Allow FILE command
Dynamic file locking	(None)	Dynamic locking *aoption* (bits 10:1)	Disallow dynamic locking
Forms-alignment message	FORMS	*formmsg*	No forms message sent
User labels for disk file	(None)	*userlabels*	No user labels processed
File labels for magnetic tape files	LABEL NOLABEL	Labeled tape *foption* (bits 6:1)	No label
File type	STD MSG CIR	File type *foption* (bits 2:3)	Standard file

To be effective, a FILE command must be issued before your file is accessed; it takes effect when the file is accessed. A FILE command remains in effect until the job or session ends, until it is cancelled 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 CARDS (indicating a card file) and then run three programs that reference DATAFL, all three programs will access the file CARDS. 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 you run two programs, both referencing a new temporary file on disk named DFILE. Before you run 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
    ⋮
```

For more information about using the FILE command, see the *MPE V Commands Reference Manual* (32033-90006).

Summary of General Rules For Overrides

If a FILE command has been entered that contradicts some of the 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:

DISC FILE LABEL

overrides

FILE COMMAND

overrides

FOPEN

overrides

FILE SYSTEM DEFAULTS

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 `FOPEN` calls cannot alter physical characteristics of an existing file, but they can alter the way the file is to be used: access parameters, whether to use buffered mode or whether to permit file locking are examples of the characteristics `FILE` and `FOPEN` can affect.

File Identification

You will probably want to identify your files in some way, so you can distinguish between them or to remind yourself of their applications. When you consider how to identify your files, both to yourself and to the system, there are several questions to bear in mind:

- Where will file identification information be stored?
- Is special non-data storage required?
- Does the file need a special file code associated with it?
- Should the file have a name?

System File Label

The system file label contains all the information about your file that you specified when you created it. Information about your file's structure, the format of its records, and details about its intended use are permanently recorded. Here once a file has been created, the system file label cannot be altered. The contents of a standard disk file label are listed in Table 3-3.

Table 3-3. Disk File Label Contents

Words	Contents
0-3	Local file name.
4-7	Group name
8-11	Account name
12-15	Identity of file creator
16-19	File lockword
20-21	File security matrix
22 (bits 0:8)	Language attribute
(bits 15:1)	File secure bit: If 1, file secured If 0, file released

Table 3-3. Disk File Label Contents (continued)

Words		Contents
23		File creation date
24		Last access date
25		Last modification date
26		File code
27		Private volume information
28	(bits 0:1)	Store bit (if on, STORE in progress)
	(bit 1:1)	Restore bit (If on, RESTORE in progress)
	(bit 2:1)	Load bit (If on, program file is loaded)
	(bit 3:1)	Exclusive bit (If on, file is opened with exclusive access)
	(bits 4:4)	Device sub-type
	(bits 8:6)	Device type
	(bits 14:1)	File open for write
	(bits 15:1)	File open for read
29	(bits 0:8)	Number of user labels written
	(bits 8:8)	Number of user labels
30-31		Maximum number of logical records
32-33		File access information (while file is open)
34		Checksum
35		Cold-load identity
36		Foptions specifications
37		Logical record size (in negative bytes)
38		Block size (in words)
39	(bits 0:8)	Sector offset to data
	(Bit 11:5)	Number of extents, minus 1
40		Last extent size
41		Extent size
42-43		Number of logical records in file
44-107		Two-word addresses of up to 32 disk extents beginning with address of first extent (words 44-45)
108-109		Restore time
110		Restore date
112-113		Start of file block number
114-115		Block number of End-of-File
116-117		Number of open and close records (MSG File)
118-119		Time last modified
124-127		Device class (in ASCII)

Non-Data Storage: User Labels

If you want some special identifying feature for your file, you can record it in a user label. For example, labels can be used on files that are frequently updated to maintain the time of the last update. These special labels can be used for files on disk or tape. If you want a user label in a tape file, the tape must be labeled with an ANSI-standard or IBM-standard label.

Note

Since labels cannot share a block with data, the first data record in a file will begin in the block following the last user label. If your blocks are large and your last user label occupies only a small part of a block, file space might be wasted.

Writing a User Label on a Disk File

When a disk file is created, MPE automatically supplies the system file label in the first sector of the first extent occupied by that file. User labels are stored just after the system file label, and will begin in the system file label's block (if it is two sectors or more). The maximum number of user-supplied labels for any file must be specified in the userlabels parameter of the FOPEN intrinsic call that creates the file; you may have a maximum of 254 user labels, each 128 words long. In Figure 3-2 the FOPEN intrinsic call:

```
DFILE2:=FOPEN(DATA2,%4,%4,128,,,1);
```

opens a new file and specifies 1 for the userlabels parameter (last parameter before parenthesis in this example), meaning that one 128-word user label will be set aside. Any attempt to write a label beyond this will result in a CCG condition code and the intrinsic request will be denied.

For example, the statement:

```
FWRITELABEL(DFILE2,LABL,9,0);
```

calls the intrinsic FWRITELABEL to write a user-supplied label. The parameters supplied in the intrinsic call are:

filenum Supplied by DFILE2, which was assigned the file number when the FOPEN intrinsic opened the file.

target The array LABL, containing the string "EMPLOYEE DATA FILE", which will be written as the user file label.

tcount 9 words, specifying the length of the string to be transferred from the array LABL (the remaining 119 words are wasted).

labelid 0, specifying the number of the label. (0=first label, 1=second label, etc.)

If the label is written successfully, a CCE condition code results; any subsequent FWRITELABEL calls specifying a previous label will overwrite it.

```
  $CONTROL
  BEGIN
     BYTE ARRAY DATA1(0:7):="DATAONE";
     BYTE ARRAY DATA2(0:7):="DATATWO";
     ARRAY LABL (0:127):="EMPLOYEE DATA FILE";
     ARRAY BUFFER (0:127);
     INTEGER DFILE1.DFILE2.DUMMY;
   ⋮
DFILE2:=FOPEN(DATA2,%4,%4,128,,,1);
   ⋮
 FWRITELABEL (DFILE2,LABL,9,0);
   ⋮
 FCLOSE(DFILE2,2,0);
   ⋮
   END.
```

Figure 3-2. FWRITELABEL Intrinsic Example (Disk)

Reading a User File Label on a Disk File

To read a user file label, you use the FREADLABEL intrinsic.

In Figure 3-3, the FOPEN intrinsic call:

```
DFILE2:=FOPEN(DATA2,%6,%4,128);
```

contains *aoptions* parameter %4, which specifies input/output access (data of the file, not the labels). The statement:

```
FREADLABEL(DFILE2,BUFFER,128,0);
```

reads a user file label from the file specified by DFILE2. The parameters specified in the intrinsic call are:

filenum Supplied by DFILE2, which was assigned the file number when the FOPEN intrinsic opened the file.

target BUFFER, the array in the stack to which the file label is transferred.

tcount 128, specifying the maximum number of words to be transferred.

labelid 0, specifying the number of the label to be read.

If the label is read, a CCE condition code results.

```
$CONTROL
BEGIN
  BYTE ARRAY DATA2(0:7):="DATATWO";
  ARRAY BUFFER (0:127)
⋮
    DFILE2:=FOPEN(DATA2,%6,%4,128);
⋮
    FREADLABEL (DFILE2,BUFFER,128,0);
⋮
END.
```

Figure 3-3. FREADLABEL Intrinsic Example (Disk)

File Codes

MPE subsystems often create special-purpose files whose functions are identified by four-digit integers called file codes, written in their system file labels. For instance, compilers create user subprogram library (USL) files, written in a special format and identified by the code 1024, for the compilation of 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 wanted 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 DESGX=DESGB;CODE=2 File code
:RUN FILEPROD
```

If you later wished to determine the classification to which this file belonged, you could use the LISTF command with an information level of 1, which would print the file name, file code, and other information about the file. The LISTF command is discussed in the *MPE V Commands Reference Manual* (32033-90006). Alternatively, you could determine the file code by calling the FGETINFO intrinsic, as discussed in the *MPE V Intrinsics Reference Manual*(32033-90007).

The file codes that have particular HP-defined meanings are listed in Table 3-4.

Note

For user files, you may use as file codes any number from 0 through 1023. Numbers from 1024 upwards are generally reserved for special system files. File codes can only be specified at the time the file is created; if you do not specify a file code when you create a file, the MPE default value of zero applies.

Table 3-4. Reserved File Codes

Mnemonic	File Code	Meaning
1024	USL	User Subprogram Library
1025	BASD	BASIC Data
1026	BASP	BASIC Program
1027	BASFP	BASIC Fast Program
1028	RL	Relocatable Library
1029	PROG	Program File
1031	SL	Segmented Library
1035	VFORM	View Form File
1036	VFAST	View Fast Forms File
1037	VREF	View Reformat File
1040	XLSAV	Cross Loader ASCII File
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 Workfile
1059	TDPXQ	TDP Workfile (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 Logfile
1100	WDOC	HPWORD Document
1101	WDICT	HPWORD Hyphenation Dictionary
1102	WCONF	HPWORD Configuration File
1103	W2601	HPWORD Attended Printer Environment
1110	PCELL	IFS/3000 Character Cell File
1111	PFORM	IFS/3000 Form File
1112	PENV	IFS/3000 Environment File
1113	PCCMP	IFS/3000 Compiled Character Cell File
1114	RASTR	Graphics Image in RASTR Format

Reserved File Codes (continued)

Mnemonic	File Code	Meaning
1130	OPTLF	OPT/3000 Logfile
1131	TEPES	TEPE/3000 Script File
1132	TEPEL	TEPE/3000 Logfile
1133	SAMPL	APS/3000 Logfile
1139	MPEDL	MPEDCP/DRP Logfile
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 Workfile
1156	DSTOR	Store File for RAPID/3000 Utility DICTDBU
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	HPDESK Distribution List
1167	MTEXT	HPDESK Text
1168	MARPA	ARPA Message File
1169	MARPD	ARPA Distribution List
1170	MCMND	HPDESK Abbreviated Commands File
1171	MFRTM	Reserved
1172		Reserved
1173	MEFT	Reserved
1174	MCRPT	Reserved
1175	MSERL	Reserved
1176	UCSF	Reserved
1177	TTYPE	Term Type File
1178	TVFC	Term Vertical Format Control File
1192	NCONF	Network Configuration File
1193	NTRAC	Network Trace File
1194	NLOG	Network Logfile
1195	MIDAS	Reserved
1211	ANODE	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

Reserved File Codes (continued)

Mnemonic	File Code	Meaning
1229	CHARD	Character Set Definition File
1230	MGCAT	Formatted Application Message Catalog
1236	BRAP	Reserved
1242	BDATA	Basic Data File
1243	BFORM	Basic Field Order File for VPLUS
1244	BSAVE	Basic Saved Program File
1245	BCNFG	Configuration File/Default Option Basic Program
1258	PFSTA	Pathflow STATIC File
1259	PFDYN	Pathflow DYNAMIC File
1270	RTDCA	Revisable Form DCA Document
1271	FFDCA	Final Form DCA Document
1272	DIU	Document Interchange Unit File
1273	PDOC	HPWORD/150 Document
1401	CWPTX	Reserved
1421	MAP	HPMAP/3000 Map Specification File
1422	GAL	Reserved
1425	TTX	Reserved
3333		Reserved

File Name

The most obvious way to identify a file is to give it a name. A file may remain unnamed, but its flexibility will be greatly limited. The FILE command cannot be used on unnamed files, and a file cannot be saved without a name. Your file's name may consist of up to eight alphanumeric characters, beginning with an alphabetic character. It may be qualified with the name of your group and account, and may have a lockword associated with it.

Formal and Actual File Designators

The name by which a program specifies 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 FILE command will reference a file by its formal designator.

Suppose that you are about to run a COBOL program named MYPROG that, in its data division, defines an input file on cards named CARDFILE. In this file, each logical record contains 80 characters and is equivalent to one block.

The coded file specification in the program appears as follows:

```
      :
DATA DIVISION.
FILE SECTION.
FD CARDFILE <------ File name

    BLOCK CONTAINS 1 RECORDS <---- Block size (1 record per block)
    DATA RECORD IS MYDATA intended for card input
    RECORDING MODE IS F <--------- Record type (fixed length)
    LABEL IS OMITTED
    RECORD CONTAINS 80 CHARACTERS.
               ▲
               |
               ----- Logical record size (80 characters)

     :
```

Although this program was designed to accept its input from punched cards, there may be occasions when you wish to read this input from a disk file. Rather than recode and recompile the program, you could reference the file in a FILE command to change the file specifications:

```
Changes file      Formal       Requests block size
specifications  designator   of 3 logical records
       ↓            ↙               ↙
      :FILE CARDFILE; REC=80, 3; DEV=DISC
      :RUN MYPROG
       ↑              ↑
Runs COBOL program     Maintains same record
                               size (80 bytes)
```

Since CARDFILE exists on DISC, its record size and blocking factor are defined by the system label and cannot be overridden by a file equation.

The formal file designator is the name by which your program specifies 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. For an old disk file, this is the file name contained in the file label and in the system or job/session temporary file directory. For a device file, it is the name optionally supplied in the DATA command and copied into the device directory. For a new disk file, it is the name you supply when you open the file; this name is then copied into the appropriate directory and entered in the file label. Whether it applies to a disk or device file, this is the name that identifies the file to the file system. It is called the actual file designator.

For example, suppose you create a file and name it DISKFILE. This name is the actual file designator. Suppose you would like to use DISKFILE in the program MYPROG described above. MYPROG specifies a file whose formal designator is CARDFILE only, so a FILE command is used to equate the formal designator to the actual designator:

```
:FILE CARDFILE=DISKFILE
          ↑         ↑
       Formal     Actual
     designator  designator
```

In this way, the FILE command provides a map between the file system's name for a file and your program's name for that file.

Renaming Your File

The creator of a file can change the name of the file, so you can rename your files. Renaming a file will change its actual file designator and its lockword, if it has one. Permanent (OLD) and temporary (TEMP) files can be renamed. Since new files do not yet exist under another name, there is no need to rename them. Changing a file's name will not change its domain.

To rename your files, use the RENAME command; it is discussed in detail in the *MPE V Commands Reference Manual* (32033-90006).

Devices and Device files

Devices required by files are allocated automatically by the file system. You can specify these devices by class (such as any card reader or line printer) or by a logical device number related to a particular device (such as a specific line printer). A unique logical device number (LDEV) is assigned to each device when the system is configured. Regardless of what device a particular file resides on, when your program requests to read that file, it references the file by its formal designator. The file system then determines the device on which the file resides, and its disk address if applicable, and accesses it for you. When your program writes information to a file destined for an output device such as a line printer, again the program refers to the file by its formal designator. The file system then automatically allocates the required device to that file. Throughout its life, every file remains device-independent, that is, it is always referenced by the same formal designator regardless of where it currently resides.

Both the device class name and the LDEV associated with a device are determined by the system supervisor or console operator when they add the device to the system. The device class name is an arbitrary name that can be allocated to more than one device. The logical device number, however, is unique for each device; it may range from 1 to 255. As an example, devices might be configured with the class names and logical device numbers shown in Table 3-5.

Table 3-5. Device Configurations

DEVICE	LDEV	Device Classname
System Disk (Required)	1	SYSDISC
Disk	2	DISC
Serial Disk	3	SDISC
Card Reader	5	CARD
Line Printer	6	LP, PRINTER
Magnetic Tape	7	TAPE, TAPE0
Magnetic Tape	8	TAPE, TAPE1
Magnetic Tape	9	TAPE, TAPE2
Magnetic Tape (Job Accepting)	10	JOBTAPE
Line Printer	11	LP
Laser Printer	14	EPOC, PP, LPS
Console	20	CONSOLE
Terminal	21	TERM
Terminal	22	TERM

In this configuration, the card reader is assigned LDEV5 and device class name CARD. In this case, you could make a unique reference to this device by using either the Ldev or the class name CARD (since no other device shares this class name) when you open the file. In the case of a magnetic tape unit, you could make specific references to LDEV 7, 8, or 9, or to the device classes TAPE0, TAPE1, or TAPE2 respectively. But if you are willing to use any magnetic tape unit, you could make a non-specific reference to the class name TAPE, which would provide the first tape unit available for your file.

When an SPL program opens a file, it can specify any device for that file in the FOPEN intrinsic. If it specifies no device, the class name DISC is assigned by default. Programs written in other languages often restrict the devices you can use for certain files. For instance, a FORTRAN program always equates the file named FTN05 to the standard input device, and the file named FTN06 to the standard listing device. In many cases, if you do not or cannot specify a device for a file in such programs, the program assumes the system default: the class name DISC.

You can, however, override the programmatic device specifications by using the FILE command to specify different devices. For example, suppose you plan to use the BASIC Interpreter from a terminal and wish to direct your program listing to any line printer rather than the subsystem default device (which is the standard listing device, your terminal). You first define the listing file, arbitrarily named PRINTER, as a line printer (class name LP) in a FILE command. After you issue the BASIC command to invoke the interpreter, you enter your BASIC program, which includes a LIST command that directs output to the file named PRINTER, which is now recognized as a line printer:

```
   :
:FILE PRINTER; DEV=LP   Defines PRINTER as a
:BASIC                  line printer file
   :
>10 FOR I=1 TO 10       Invokes BASIC Interpreter
   :
>LIST,OUT=PRINTER       Transmits output to PRINTER
   :
```

The BASIC command is discussed in the *MPE V Commands Reference Manual* (32033-90006).

If a file is a spooled device file, you can assign an output priority to the file. The priority can range from 1 (lowest) to 13 (highest). The console operator will establish the outfence to limit spooling activity. Spooled output files with priorities lower than or equal to the outfence are not printed or punched until the outfence is lowered or the priorities are raised by the console operator. Suppose you are running a program that will print an extensive output file at a time when the computer is left unattended. To safeguard against problems arising from the printer jamming or running out of paper while it prints the file, you could specify an output priority less than the current outfence (8), and request the operator to lower the outfence upon return to the machine room. When this is done, your file can be transmitted from disk to printer. You might specify the priority as follows:

```
                          Output priority
                          ↙
:FILE LONGFILE; DEV=LP,6
:RUN PROGX
```

Device-Dependent Characteristics

Certain file characteristics for device files are restricted by the devices on which the files reside. For instance, the file system always assigns a blocking factor of 1 to any file read from a card reader regardless of the blocking factor specified in your FOPEN call or FILE command. Device-dependent restrictions are summarized in Figure 3-4.

INPUT ONLY DEVICES (SERIAL)

- Card Reader/Paper Tape Reader
  - No carriage control
  - Undefined-length records
  - If card reader, ASCII only (can only read ASCII cards using FCONTROL)
  - Blocking factor = 1
  - Domain = 1 (OLD permanent)
  - If not ASCII, then NOBUF
  - If access type = 1,2,3 then access violation results

INPUT/OUTPUT DEVICES (PARALLEL)

- Terminals
  - ASCII
  - NOBUF
  - Undefined-length records
  - Blocking factor = 1

INPUT/OUTPUT DEVICES (SERIAL)

- Magnetic Tape Drive
- Serial Disk Drive
  - No restriction

OUTPUT ONLY (SERIAL)

- Line Printer/Card Punch/Paper Tape Punch/Plotter
  - If Paper Tape Punch, ASCII only
  - Undefined-length records
  - Blocking factor = 1
  - Domain = NEW
  - Access Type = 1, write only (if read only specified, accesqs violation results)
- Laser Printer
  - Initially and always spooled
  - Write only access
  - All other restrictions same as for line printer

UNDEFINED (COMMON CHECKING)

- If carriage control specified and not ASCII, access violation results

Figure 3-4. Device-Dependent Restrictions

Headers and Trailers

A facility for printing header and trailer records can be enabled by the Console Operator through the Console command **HEADON**. When this facility is enabled and an output device file is directed to a card punch, the file system automatically punches a header card and a trailer card identifying the job that produced the file. If an output device file is directed to a line printer, the file system automatically prints header and trailer pages identifying the job that produced the file. The console operator can disable the header facility by entering the **HEADOFF** command.

Special Forms

When a program opens a new output device file, it may request special forms. This request transmits a user forms message to the Operator's console, along with a request to mount the forms. The Operator may respond as follows:

1. If the program specified a device class name for the file, the Operator may allocate any unowned device in the class.
2. If the program specified a particular LDEVfor the file, the file system asks the Operator to mount the forms on the device requested if it is available.

When the Operator allocates a line printer, the file system initiates a dialog to align the forms. A standard record of the following form is output to the line printer:

```
                                                  Column 132
     •           •           •          •           ↙

.... .... . 1.... .... . 2.... .... . 3.... .... 3 ..
```

This record is followed by a Console message which asks the Operator if the forms are properly aligned. This transaction is repeated until the Operator indicates proper alignment. Now the file can be output.

When a program closes a device file with special forms, the file system notifies the Operator that the forms are no longer needed on the device.

If special forms are mounted on a device and a device file not requiring them is assigned to the device, the file system automatically asks the Operator to mount standard forms or paper.

Foreign Disk Facility.

The foreign disk facility (FDF) allows you to use the file system to access and alter disk packs and flexible diskettes that do not have standard HP 3000 file system disk label formats. When mounted, a disk volume with an unrecognizable disk label is assumed to be a foreign disk. Disks and diskettes must be physically compatible with HP hardware. For example, the IBM 3741 format diskettes (64 words per sector), are compatible.

When using the FOPEN intrinsic to open a foreign disk file, the *recsize* is forced to 128 words (IBM diskettes are forced to 64 words). The file system will treat disk sectors as file records, allowing you to manipulate the foreign file as if it were an MPE created file.

Section Divider

4. Domains

4

Domains

The various ways to classify a file is, permanent or temporary, or it may exist only to one particular process. This is referred to as a domain. The file system maintains separate directories to record the location of temporary (or TEMP) files and permanent (or OLD) files. Of course, there is no file system directory for files which 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?

TYPES OF DOMAINS

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 it 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 will vanish, unless you take actions to retain it.

TEMP Files

A TEMP file is one which already exists, but which is known only to the job or session which created it. Some or all of the space for a TEMP 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. It will be discarded when its creating job or session is over.

OLD Files

An OLD file exists as a permanent file in the system. 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 an OLD file has already been allocated, and its physical characteristics have been defined.

The features of NEW, TEMP and OLD files are listed in Table 4-1.

Table 4-1. Features of NEW, TEMP, and OLD Files

NEW Files	TEMP Files	OLD 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.

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 permitted domains are summarized in Table 4-2.

Table 4-2. File Domains Permitted

Device Type	Domain
Disk	NEW, OLD, or TEMP
Card Reader	OLD
Paper Tape Reader	OLD
Terminals	NEW or OLD
Printing Reader/Punch	NEW or OLD
Synchronous Single-Line Controller	NEW or OLD
Programmable Controller	NEW or OLD
Magnetic Tape Unit	NEW or OLD
Line Printer	NEW
Paper Tape Punch	NEW
Plotter	NEW

Changing Domains

A file need not always stay in the same domain. Any file can be made permanent, or it 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 the disposition of the file when it is closed. For details about how the FCLOSE intrinsic handles file domain disposition, see the *MPE V Intrinsics Reference Manual* (32033-90007).

A file in any domain may be deleted if the DEL parameter is used in a file equation. For example, suppose you have an old file named OLDFL, and wish 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 will be 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 will maintain 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 old file in the system, enter:

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

TEMPFL will be 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 are being accessed and closed by the program.

By using the MPE 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 will immediately reclassify it as a permanent file. If there were a lockword associated with TEMPDATA, you would be prompted to enter 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 you choose.

For more information about the FILE and SAVE commands, consult the *MPE V Commands Reference Manual* (32033-90006).

Directory Search

There are two directories with addresses of files: the job temporary file directory (JTFD) for the addresses of temporary files and the System File Directory for the addresses of permanent files. There is no directory for new files. When both directories are searched for a file address, the JTFD is searched first. There is one JTFD for every job/session; each is kept in an extra data segment.

Listing Files

To obtain a list of your permanent files, use the LISTF command. Use the LISTFTEMP command to list your temporary files and FILE equations. The LISTF and LISTFTEMP commands are discussed in detail in the *MPE V Commands Reference Manual* (32033-90006).

Section Divider

5. File Operation

5

File Operation

File Operation

In this chapter, we will explore the operation and usage of files. As you read this chapter, keep these considerations in mind:

- How will the file be referenced?
- How will the file be used?
- Will others be allowed concurrent access?
- Will the concurrent access need special management?
- Are there special features required to access the file?

Specifying 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. 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 1 to 8 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 file reference format, as follows:

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

The file reference format cannot exceed 35 characters, including delimiters.

When you reference a file that belongs to your logon account and group, you need only use the file reference format in its simplest form, which includes only a file name that may range from 1 to 8 alphanumeric characters, beginning with a letter. In the following examples, both formal and actual designators appear in this format:

```
    Formal        Actual
   designator   designator
   ---| ----   ----|------
      |      _|
      |       |
      |       |
      V       V
FILE ALPHA=BETA
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 will recognize 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 logon, 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 public group of the System Account. This is a special account available to all users on every system, 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 file reference 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)
```

Another example, suppose you are logged on under the group named 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 file reference 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 GENentity=GENINFO.PUB.SYS
```

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 logon 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 file reference, the system will supply 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 file reference 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 file reference parameter of the BUILD command or the formal designator parameter of the 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 command or 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 as follows:

- In batch mode, supply the lockword as part of the file designator (file reference format) specified in the FILE command or 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 1 to 8 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 will be 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 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 logon. Three examples of FILE commands referencing lockwords are shown below; the last command illustrates the complete, fully qualified form of the file reference format:

```
:FILE Aentity=GOFILE/Z22 <----------------  Lockword

                         |-------------------  Lockword
                         |
                         V
:FILE Bentity=FILEM/LOCKB.GR07

                            |-----------------  Lockword
                            |
                            V
:FILE Centity=PAYROLL/X229AD.GROPN.ACCT10
```

A file may have only one lockword at a time. You can change the lockword by using the RENAME command or the FRENAME intrinsic. Both are discussed later in this chapter. You can also initially assign a lockword to an existing file with this command or intrinsic. To do either of these tasks, you must be the creator of the file.

Back Referencing 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 respecify all those parameters 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`	Back references specifications for FILEA
`RUN PROGB`	Runs program B

This technique is called back referencing 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.

Generic Names

The commands LISTF, LISTVS, REPORT, RESTORE, and STORE permit the specification of sets of files, volume set definitions, or groups. For example, a fileset for the STORE command can be specified in the form:

filedesignator [. groupdesignator [. acctdesignator]]

The characters @, #, and ? can be used as wild card characters.

These wild card characters have the following meanings:

@ - specifies zero or more alphanumeric characters.
- specifies one numeric character.
? - specifies one alphanumeric character.

The characters can be used as in the following examples:

n@ Refers to all files starting with the character *n* .

@n Refers to all files ending with the character*n* .

n@x Refers to all files starting with the character *n* and ending with the character x.

n## ... # Refers to all files starting with the character n followed by up to seven digits.

?n@ Refers to all files whose second character is *n* .

n? Refers to all two-character files starting with *n* .

?n Refers to all two-character files ending with*n* .

System-Defined Files

System-defined file designators indicate files that the file system uniquely identifies as standard input/output devices for jobs and sessions. These designators are described in Table 5-1. When you reference them, you use only the file name; group or account names and lockwords do not apply.

Table 5-1. 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, card reader, 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 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 non-existent 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 but no physical output is actually done. Thus, $NULL can be used to discard unecessary data 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 INPUT and directs output to a file programmatically defined as OUTPUT. Your program specifies that these are disk files, but you wish to respecify these files so that INPUT is read from the standard input device and OUTPUT is sent to the standard listing device. You could enter the following commands:

```
:FILE INPUT=$STDIN
:FILE OUTPUT=$STDLIST
:RUN MYPROG
```

Input/Output Sets

All file designators can be classified as those used for input files (Input Set) and those used for output files (Output Set). For your convenience, these sets are summarized in Table 5-2 and Table 5-3.

Table 5-2. 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.
formal designator	Back reference to a previously defined file.
file reference	File name, and perhaps account and group names and lockword. Indicates an old file. May be a job/session temporary file created in this or a previous program in current job/session, or a permanent file saved by any program or a `BUILD` or `SAVE` command in any job/session.

Table 5-3. Output Set

File Designator	Function/Meaning
`$STDLIST`	Job/session list device.
`$OLDPASS`	Last file passed. Discussed in the following pages.
`$NEWPASS`	New temporary file to be passed. Discussed in the following pages.
`$NULL`	Constantly empty file that returns end-of-file indication when written.
formal designator	Back reference to a previously defined file.
file reference	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 job/session 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.

Determining Interactive and Duplicative File Pairs

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 a programmed 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 card reader 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 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 FOPEN intrinsic opened the files.

A word is returned to ABLE showing whether the files are interactive or duplicative. The word returned contains two significant bits, 0 and 1:

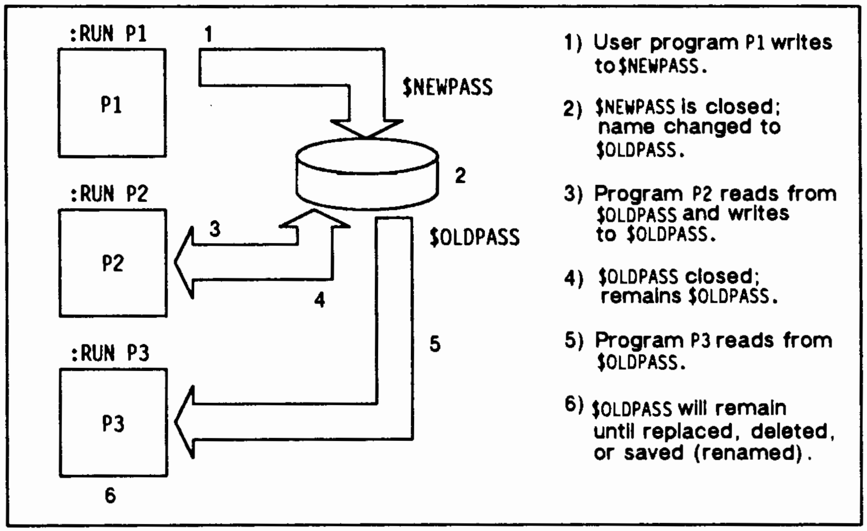
If bit 15 = 1, INFILE and LISTFILE form an interactive pair.
If bit 15 = 0, INFILE and LISTFILE do not form an interactive pair.
If bit 0 = 1, INFILE and LISTFILE form a duplicative pair.
If bit 0 = 0, INFILE and LISTFILE do not form a duplicative pair.

Passed Files

Programmers, who write compilers or other subsystems, sometimes create a temporary disk file that can be automatically passed to succeeding MPE commands within a job or session. This file is always created under the special name $NEWPASS.

When your program closes the file, MPE 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 5-1.



1) User program P1 writes to $NEWPASS.

2) $NEWPASS is closed; name changed to $OLDPASS.

3) Program P2 reads from $OLDPASS and writes to $OLDPASS.

4) $OLDPASS closed; remains $OLDPASS.

5) Program P3 reads from $OLDPASS.

6) $OLDPASS will remain until replaced, deleted, or saved (renamed).

LG200016_018

Figure 5-1. Passing Files Between Program Runs

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 by the program in PROG1 by the name INTERFIL.) When PROG2 is run, it receives $NEWPASS (now known by the actual designator $OLDPASS), referencing that file as SOURCE2. Note that only one file can be designated for passing:

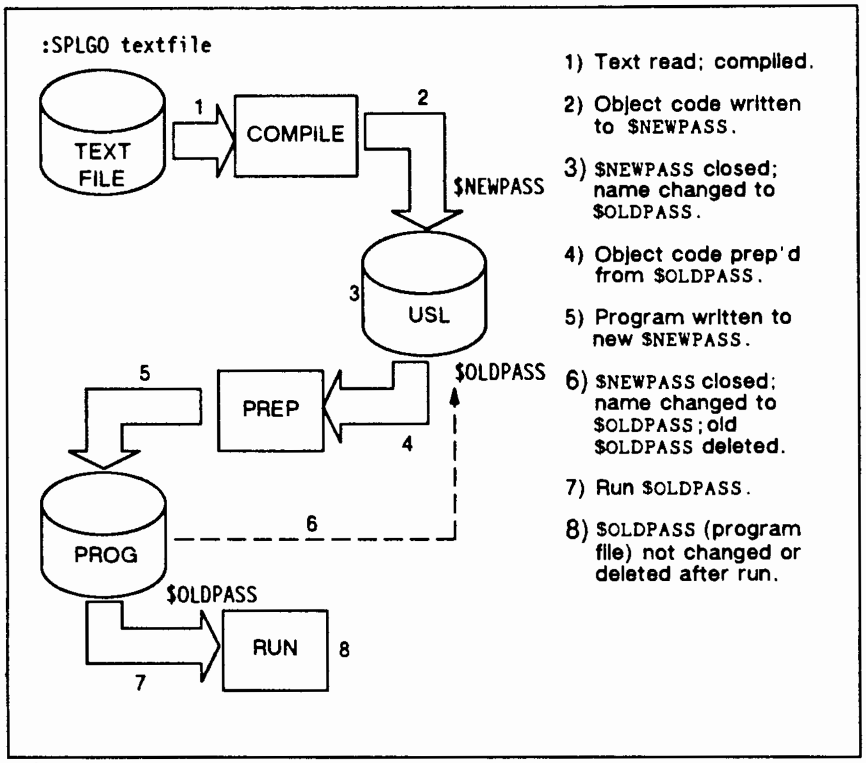
```
⋮
:FILE SOURCE1=DSFILE
:FILE INTERFIL=$NEWPASS <---------|   Same file
:RUN PROG1
:FILE SOURCE2=$OLDPASS <---------|    Same file
:RUN PROG2
⋮
```

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 prepared 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.

This file is prepared for execution. The prepared 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. The $OLDPASS file 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 shown in Figure 5-2.



1) Text read; compiled.

2) Object code written to $NEWPASS.

3) $NEWPASS closed; name changed to $OLDPASS.

4) Object code prep'd from $OLDPASS.

5) Program written to new $NEWPASS.

6) $NEWPASS closed; name changed to $OLDPASS; old $OLDPASS deleted.

7) Run $OLDPASS.

8) $OLDPASS (program file) not changed or deleted after run.

LG200016_019

Figure 5-2. Passing Files Within a Program Run

Comparing $NEWPASS and $OLDPASS to Other Disk Files

$NEWPASS and $OLDPASS are specialized disk files with many similarities to other disk files. Cpmparisons of $NEWPASS to new files, and $OLDPASS to old files, are given in Table 5-4 and Table 5-5.

Table 5-4. 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 job/session table. (Job Information Table). Delete control block entry.
Disk address not saved (Not in any directory).	Disk address saved for future use in the job/session.

Table 5-5. 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 job/session.

Shared File Considerations

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.

Simultaneous Access of Files

When an FOPEN request is issued for a file, that request is regarded as an individual accessor of the file and a unique file number, set of buffers, and other file control information is established for that file. Even when the same program issues several different FOPEN calls for the same file, each call is treated as a separate accessor.

Under the normal (default) security provisions of the system, 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 FOPEN intrinsic calls. Users running those programs can, in turn, override both the defaults and program options through the FILE command. The options are listed in Table 5-6. The actions taken by the system when these options are in effect and simultaneous access is attempted by other FOPEN calls are summarized in Table 5-7. The action taken depends upon the current use of the file versus the access requested.

Table 5-6. File Sharing Restriction Options

ACCESS RESTRICTION	:FILE PARAMETER	DESCRIPTION
Exclusive Access	EXC	After file is opened, prohibits concurrent access in any mode through another 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 FOPEN request, whether issued by this or another program, until this program issues FCLOSE or terminates.
Sharable Access	SHR	After file is opened, permits concurrent access to file in any mode through another FOPEN request issued by this or another program, in this or any session or job. Each accessor uses copy portion of file within its own buffer.

Table 5-7. Actions Resulting from Multi-Access of Files

	REQUESTED ACCESS GRANTED, UNLESS NOTED						
	Current Use	FOPEN for Input		FOPEN for Output		FOPEN for Input/Output	
Requested Access		SHR/ MULTI/ GMULTI	SEMI	SHR/ MULTI/ GMULTI	SEMI	SHR/ MULTI/ GMULTI	SEMI
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
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
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

Note

In all cases, when the first accessor to a file opens it with exclusive (EXC) access, all other attempts to open the file will fail.

If input or output access is requested, and input only is obtained, examine the file options after a successful `FOPEN` to see if you have obtained a needed output access. See `FGETINFO` and `FFILEINFO`.

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 program 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
```

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 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
```

Share Access

When opened with the share option, a file can be shared (in all access modes) among several FOPEN requests, whether they are issued from the same program, different programs within the same job/session, or programs running under different jobs/ sessions. Each accessor transfers its input/output to and from the file via 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 multi-access and/or locking 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 share access
RUN RDPROG
```

Multi-Access

This option extends the features of the share access option to allow a deeper level of multiple access. Multi-access not only makes the file available simultaneously to other accessors (in the same job/session), but permits them to use the same buffers, blocking factor, and other file-control information. The file must be buffered. Multi-access may not be used on files that are opened with the NOBUF Option. Thus, transfers to and from the file occur in the order they are requested, regardless of which program in your job/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

Multi-access allows the file to be shared (in all access modes) among several FOPEN requests from the same program, or from different concurrently running programs in the same job/session. Unlike share access, however, multiaccess does not permit the file to be shared among different sessions and jobs.

Global Multi-Access

This option extends the features of the multi-access option to permit simultaneous access of a file by processes in different jobs/sessions. As in multi-access, accessors use the same buffers, 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 will maintain separate buffers and pointers.

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 will remain in effect until the user that requested it closes the file.

Sharing the File

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) will not cause physical I/O to occur until a block is filled, which typically will contain 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 restricted on its FLOCK call until the reader signals that it is done by calling FUNLOCK. If the reader should execute while the writer is performing a write, the reader will be restricted 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 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. If any of the other files have buffered access, it is necessary to empty the buffers for each file to allow both readers and writers access to clean copies. This can be done by executing FCONTROL 2 (complete input/output) for each file prior to executing FUNLOCK. If posting the current record pointer as the END-OF-FILE is needed, use FCONTROL 6 (write end of file) instead of FCONTROL 2.

For more information about the FLOCK and FUNLOCK intrinsics, consult the *MPE V Intrinsics Reference Manual* (32033-90007).

Section Divider

6. Data Transfer

6

Data Transfer

This chapter examines how the file system transfer of data. As you read this chapter, keep these considerations in mind:

- How are records selected for transfer?
- What intrinsics are used for data transfer?
- Will there be any file buffering?
- If so, how many buffers will be used?

Record Pointers

The file system uses record pointers to find specific records for your use. *Physical record pointers* are used to locate specific blocks on disk; *logical record pointers* are used to block and deblock the logical records in a physical record and indicate specific logical records within a file buffer. (NOBUF files have physical record pointers only.

Figure 6-1 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.

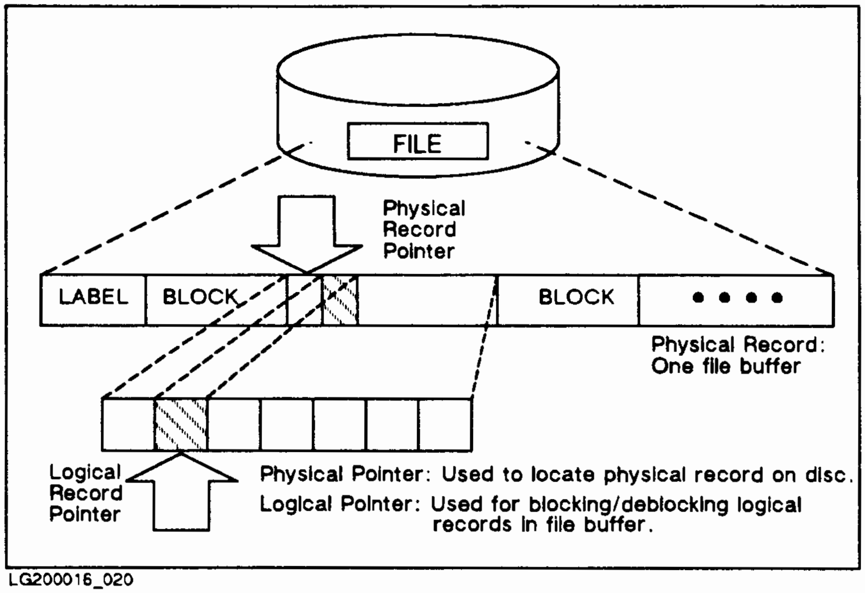

FILE
Physical
Record
Pointer
LABEL
BLOCK
BLOCK
● ● ● ●
Physical Record:
One file buffer
Logical
Record
Pointer
Physical Pointer: Used to locate physical record on disc.
Logical Pointer: Used for blocking/deblocking logical records in file buffer.
LG200016_020

Figure 6-1. Record Pointers

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.

Pointer Initialization

When you open your file, the 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, however, the record pointer will be moved to the end of the file prior to any write operation. This will ensure that any data you write to the file will be added to the end of the file rather than written over existing data.

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

Various file system intrinsics are designed to move records to and from your file, but how do they choose the records they want? The record pointer for a particular file indicates the specific location where that 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 you wish to access.

There are three methods of record selection: the default method, in which you transfer data to or from the place which the record pointer currently indicates; random access, in which you move the record pointer before transferring data; and update selection, in which you choose a record and write a new record over it.

Default Record Selection

When you use this method of record selection, you assume 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 next record. For this reason, this method is also called sequential 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 and adds that to the record pointer.

Random Access

If the record pointer is not indicating the location you want, you can use the random access method to move the pointer and begin your transfer wherever you like. This method is 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. FREADDIR and FWRITEDIR may be issued only for a disk file composed of fixed-length or undefined-length records.

Note

The FREADDIR and FWRITEDIR intrinsics operate in the usual manner to access foreign disks. However, on IBM diskettes sectors are numbered starting with one rather than zero, and the diskette driver adds one to all sector addresses for IBM diskettes. Therefore, you specify record number zero to access sector number one on an IBM diskette.

Figure 6-2 contains a program that reads every other record in a disk file using the FREADDIR intrinsic. The FREADDIR intrinsic call:

```
FREADDIR (DFILE2,BUFFER,128,REC);
```

reads a record from the file designated by DFILE2 (the file number was assigned to DFILE2 when the FOPEN intrinsic opened the file) and transfers this record to the array BUFFER in the stack. Up to 128 words are read from the record. The parameter REC specifies which record is read. The double integer value ' ' (double integers are indicated by the suffix D in SPL) was assigned to REC in statement number 9, and so the first time the LIST'LOOP is executed, the first record in the file (logical record number 0) is read. REC is incremented by 2D each time the loop is executed, so the third logical record (logical record number 2) is read the second time the loop is executed, then the fifth, seventh and so on. The record pointer is advanced by one each time the FREADDIR intrinsic is executed.

The record number to be read is specified by REC. The FREADDIR intrinsic does not necessarily read records in sequential order, as does the FREAD intrinsic.

If the information is not read successfully by the FREADDIR intrinsic, a CCL condition is returned. The statement:

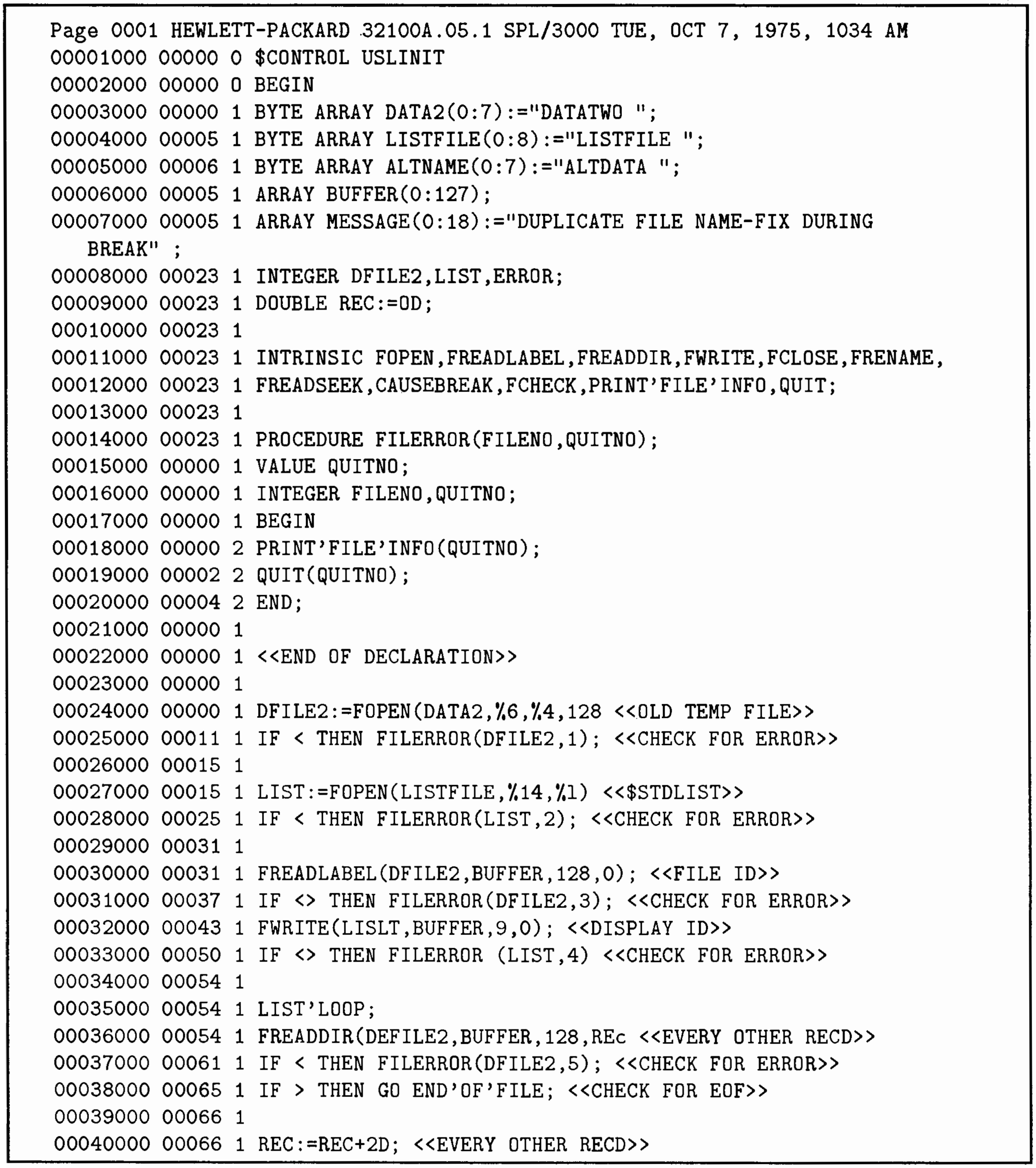
```
IF < THEN FILERROR (DFILE2,3);
```

checks the condition code and, if it is CCL, calls the error-check procedure FILERROR. The FILERROR procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FREADDIR, then aborts the program's process.

A condition code of CCG signifies an end-of-file condition and the statement:

```
IF > THEN GO END'OF'FILE;
```

transfers program control to the label END'OF'FILE when the end-of-file condition is encountered.



```
Page 0001 HEWLETT-PACKARD 32100A.05.1 SPL/3000 TUE, OCT 7, 1975, 1034 AM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00004000 00005 1 BYTE ARRAY LISTFILE(0:8):="LISTFILE ";
00005000 00006 1 BYTE ARRAY ALTNAME(0:7):="ALTDATA ";
00006000 00005 1 ARRAY BUFFER(0:127);
00007000 00005 1 ARRAY MESSAGE(0:18):="DUPLICATE FILE NAME-FIX DURING
   BREAK" ;
00008000 00023 1 INTEGER DFILE2,LIST,ERROR;
00009000 00023 1 DOUBLE REC:=0D;
00010000 00023 1
00011000 00023 1 INTRINSIC FOPEN,FREADLABEL,FREADDIR,FWRITE,FCLOSE,FRENAME,
00012000 00023 1 FREADSEEK,CAUSEBREAK,FCHECK,PRINT'FILE'INFO,QUIT;
00013000 00023 1
00014000 00023 1 PROCEDURE FILERROR(FILENO,QUITNO);
00015000 00000 1 VALUE QUITNO;
00016000 00000 1 INTEGER FILENO,QUITNO;
00017000 00000 1 BEGIN
00018000 00000 2 PRINT'FILE'INFO(QUITNO);
00019000 00002 2 QUIT(QUITNO);
00020000 00004 2 END;
00021000 00000 1
00022000 00000 1 <<END OF DECLARATION>>
00023000 00000 1
00024000 00000 1 DFILE2:=FOPEN(DATA2,%6,%4,128 <<OLD TEMP FILE>>
00025000 00011 1 IF < THEN FILERROR(DFILE2,1); <<CHECK FOR ERROR>>
00026000 00015 1
00027000 00015 1 LIST:=FOPEN(LISTFILE,%14,%1) <<$STDLIST>>
00028000 00025 1 IF < THEN FILERROR(LIST,2); <<CHECK FOR ERROR>>
00029000 00031 1
00030000 00031 1 FREADLABEL(DFILE2,BUFFER,128,0); <<FILE ID>>
00031000 00037 1 IF <> THEN FILERROR(DFILE2,3); <<CHECK FOR ERROR>>
00032000 00043 1 FWRITE(LISLT,BUFFER,9,0); <<DISPLAY ID>>
00033000 00050 1 IF <> THEN FILERROR (LIST,4) <<CHECK FOR ERROR>>
00034000 00054 1
00035000 00054 1 LIST'LOOP;
00036000 00054 1 FREADDIR(DEFILE2,BUFFER,128,REc <<EVERY OTHER RECD>>
00037000 00061 1 IF < THEN FILERROR(DFILE2,5); <<CHECK FOR ERROR>>
00038000 00065 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00039000 00066 1
00040000 00066 1 REC:=REC+2D; <<EVERY OTHER RECD>>
```

Figure 6-2. FREADDIR and FREADSEEK Example (1 of 2)

```
00041000 00072 1 FREADSEEK(DFILE2,REC); <<FILL SYSTEM BUFFER>>
00042000 00075 1 IF < THEN FILERROR(DFILE2,5); <<CHECK FOR ERROR>>
00043000 00101 1
00044000 00101 1 FWRITE(LIST,BUFFER,35,0); <<ALTERNATE RECORD>>
00045000 00106 1 IF <> THEN FILERROR(LIST7); <<CHECK FOR ERROR>>
00046000 00112 1
00047000 00112 1 GO LIST'LOOP; <<CONTINUE LISTING>>
00048000 00117 1
00049000 00117 1 END'OF'FILE:
00050000 00117 1 FCLOSE(DFILE2,1,0); <<MAKE PERMANENT>>
00051000 00123 1 IF = THEN GO DONE; <<LISTING DONE>>
00052000 00124 1 FCHECK(DFILE2,ERROR); <<FCLOSE ERROR>>
00053000 00131 1 IF ERROR=100 THEN <<DUPLICATE FILE NAME>>
00054000 00134 1 BEGIN
00055000 00134 1 FRENAME(DFILE2,ALTNAME); <<CHANGE FILE NAME>>
00056000 00137 2 CLOSE:
00057000 00137 2 FCLOSE(DFILE2,1,0); <<TRY AGAIN>>
00058000 00143 2 IF = THEN GO DONE; <<GOOD FCLOSE>>
00059000 00144 2 PRINT'FILE'INFO(DFILE2); <<PRINT ERROR>>
00060000 00146 2 FWRITE(LI ST, MESSAGE,19,0) ; <<SEEK HELP>>
00061000 00153 2 CAUSEBREAK; <<SESSION BREAK>>
00062000 00154 2 GO CLOSE; <<LOOP BACK>>
00063000 00155 2 END;
00064000 00155 1 DONE:END.
  PRIMARY DB STORAGE=%012; SECONDARY DB STORAGE=%00240
  NO. ERRORS=000; NO. WARNINGS=000
  PROCESSOR TIME=0:00:)04; ELAPSED TIME=0:00:58
```

Figure 6-2. FREADDIR and FREADSEEK Example (2 of 2)

Figure 6-3 contains a program that reads records from one file and writes these records, in inverse order, into a second file using the FWRITEDIR intrinsic. The FGETINFO intrinsic (see Appendix B, "Status Information") is used to locate the end-of-file in the file to be read. This information is returned to the variable REC.

The FREAD statement:

```
DUMMY := FREAD (DFILE1,BUFFER,128);
```

reads up to 128 words from the first record of the file DATAONE (specified by the file number assigned to DFILE1 by the FOPEN intrinsic when the file was opened) and transfers this information to the array BUFFER.

The statement:

```
REC := REC-1D;
```

decrements REC by the double integer value 1D to arrive at the logical record number of the last record in the file. Note that REC contains a current value of the last logical record + 1D as a result of the FGETINFO intrinsic call.

The FWRITEDIR statement:

```
FWRITEDIR (DFILE2,BUFFER,128,REC);
```

writes the record contained in the array BUFFER to the file specified by DFILE2. Up to 128 words are written to the record. The record is written to the location specified by REC, which contains the logical record number of the last record in the file.

If the FWRITEDIR request is successful, a CCE condition is returned. The statement:

```
IF <> THEN FILERROR (DFILE2,6);
```

checks for a "not equal" condition code and, if such a condition code is returned, the error-check procedure FILERROR is called.

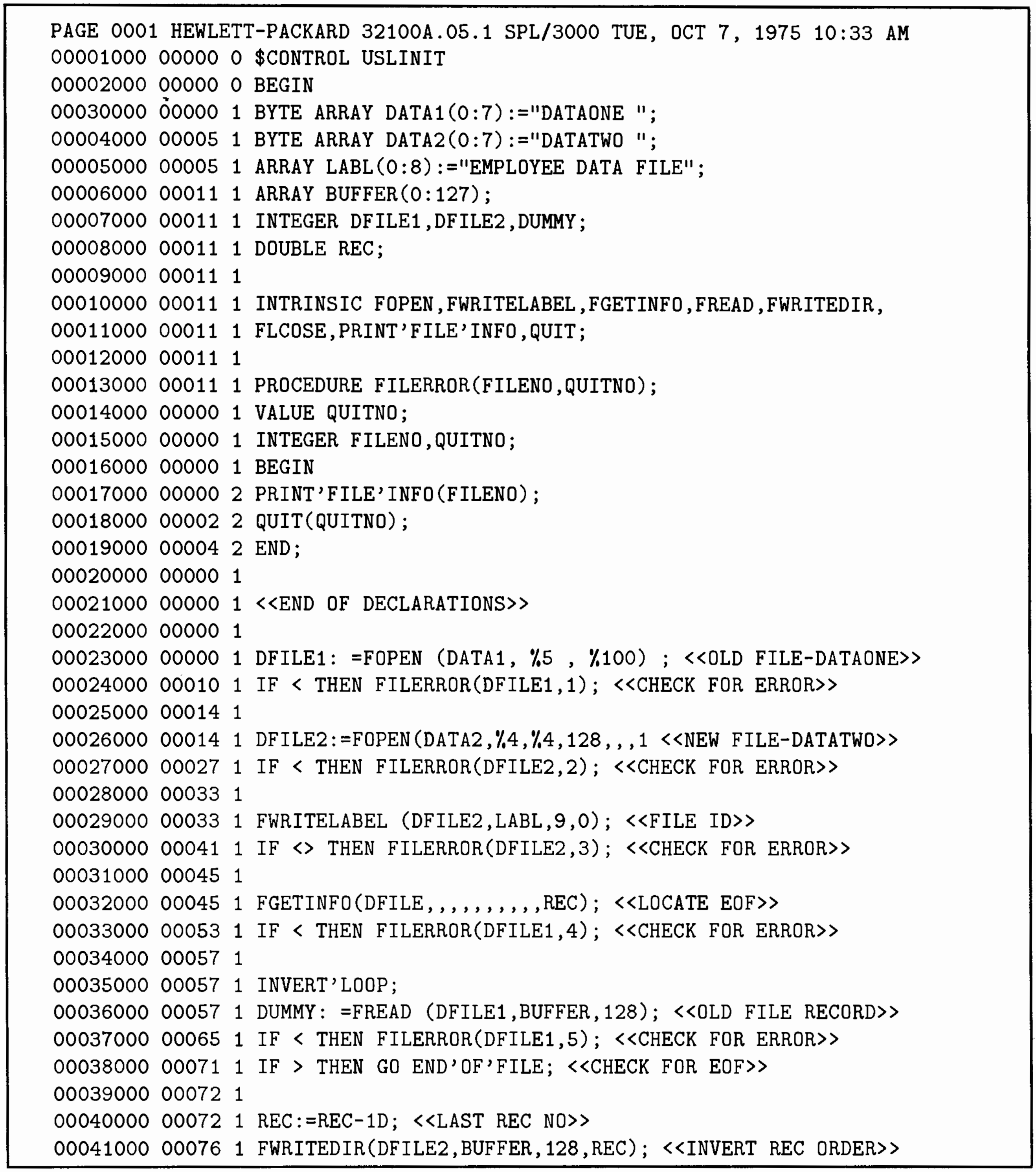
The FILERROR procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FWRITEDIR, then aborts the program's process.

If a condition code of CCE is returned, the FILERROR procedure is not executed and the:

```
GO INVERT'LOOP;
```

statement transfers program control to the statement label INVERT'LOOP, causing the invert loop to be repeated.

The second time the loop is executed, the FREAD intrinsic reads the second record from DATAONE and the FWRITEDIR intrinsic writes this record into the next-to-last record in DATATWO (REC has been decremented again by 1D). The loop repeats until the last record is read from DATAONE.



```
PAGE 0001 HEWLETT-PACKARD 32100A.05.1 SPL/3000 TUE, OCT 7, 1975 10:33 AM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00030000 00000 1 BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000 00005 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00005000 00005 1 ARRAY LABL(0:8):="EMPLOYEE DATA FILE";
00006000 00011 1 ARRAY BUFFER(0:127);
00007000 00011 1 INTEGER DFILE1,DFILE2,DUMMY;
00008000 00011 1 DOUBLE REC;
00009000 00011 1
00010000 00011 1 INTRINSIC FOPEN,FWRITELABEL,FGETINFO,FREAD,FWRITEDIR,
00011000 00011 1 FLCOSE,PRINT'FILE'INFO,QUIT;
00012000 00011 1
00013000 00011 1 PROCEDURE FILERROR(FILENO,QUITNO);
00014000 00000 1 VALUE QUITNO;
00015000 00000 1 INTEGER FILENO,QUITNO;
00016000 00000 1 BEGIN
00017000 00000 2 PRINT'FILE'INFO(FILENO);
00018000 00002 2 QUIT(QUITNO);
00019000 00004 2 END;
00020000 00000 1
00021000 00000 1 <<END OF DECLARATIONS>>
00022000 00000 1
00023000 00000 1 DFILE1: =FOPEN (DATA1, %5 , %100) ; <<OLD FILE-DATAONE>>
00024000 00010 1 IF < THEN FILERROR(DFILE1,1); <<CHECK FOR ERROR>>
00025000 00014 1
00026000 00014 1 DFILE2:=FOPEN(DATA2,%4,%4,128,,,1 <<NEW FILE-DATATWO>>
00027000 00027 1 IF < THEN FILERROR(DFILE2,2); <<CHECK FOR ERROR>>
00028000 00033 1
00029000 00033 1 FWRITELABEL (DFILE2,LABL,9,0); <<FILE ID>>
00030000 00041 1 IF <> THEN FILERROR(DFILE2,3); <<CHECK FOR ERROR>>
00031000 00045 1
00032000 00045 1 FGETINFO(DFILE,,,,,,,,,,REC); <<LOCATE EOF>>
00033000 00053 1 IF < THEN FILERROR(DFILE1,4); <<CHECK FOR ERROR>>
00034000 00057 1
00035000 00057 1 INVERT'LOOP;
00036000 00057 1 DUMMY: =FREAD (DFILE1,BUFFER,128); <<OLD FILE RECORD>>
00037000 00065 1 IF < THEN FILERROR(DFILE1,5); <<CHECK FOR ERROR>>
00038000 00071 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00039000 00072 1
00040000 00072 1 REC:=REC-1D; <<LAST REC NO>>
00041000 00076 1 FWRITEDIR(DFILE2,BUFFER,128,REC); <<INVERT REC ORDER>>
```

Figure 6-3. FWRITEDIR Example (1 of 2)

```
00042000 00103 1 IF <> THEN FILERROR(DFILE2,6); <<CHECK FOR ERROR>>
00043000 00107 1
00044000 00107 1 GO INVERT'LOOP; <<CONTINUE OPERATION>>
00045000 00116 1
00046000 00116 1 END'OF'FILE:
00047000 00116 1 FCLOSE(DFILE2,2,0); <<SAVE NEW AS TEMP>>
00048000 00122 1 IF < THEN FILERROR(DFILE2,7); <<CHECK FOR ERROR>>
00049000 00126 1
00050000 00126 1 FCLOSE(DFILE1,4,0); <<DELETE OLD FILE>>
00051000 00132 1 IF < THEN FILERROR(DFILE1,8); <<CHECK FOR ERROR>>
00052000 00136 1 END.
   PRIMARY DB STORAGE=%011; SECONDARY DB STORAGE=%00221
   NO. ERRORS=000; NO. WARNINGS=000
   PROCESSOR TIME=0:00:04; ELAPSED TIME=0:00:59
```

Figure 6-3. FWRITE Example (2 of 2)

Optimizing Direct-Access File Reading

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

The FREADSEEK intrinsic moves the record from the file to a file system buffer. Then, when the FREADDIR intrinsic call is issued, the record is transferred from this buffer to the buffer in the stack specified by FREADDIR. The use of FREADSEEK enhances the I/O process, because the file system buffer already contains the record to be read before the FREADDIR call is issued.

The LIST'LOOP in Figure 6-2 performs the following functions:

1. Issues an FREADDIR intrinsic call to transfer a record (specified by REC) from a file (specified by DFILE2) to an array (BUFFER) in the stack.
2. Increments REC by 2D.
3. Issues an FREADSEEK intrinsic call to read the record specified by the new value of REC and to transfer this record to a file system buffer.
4. Lists the record in the stack array (BUFFER) on the standard list device.
5. Repeats the loop.

The next time LIST'LOOP is executed, the FREADDIR intrinsic reads the record from the file system buffer to the stack array (BUFFER), eliminating the need for file access and thus reducing the execution time of the loop.

Update Selection

To update a logical record of a disk file, use the FUPDATE intrinsic. 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. The file system assumes the record to be updated has just been accessed in some way.

The file containing the record to be updated must have been opened with the update *aoption* specified in the FOPEN call and must not contain variable-length records. FUPDATE operates in the usual manner to update a foreign disk file. Figure 6-4 contains a program that opens an old disk file and updates records in the file. The update information (employee number) is entered from a terminal (the program is run interactively) into a buffer in the stack, then the contents of the buffer are used to update the record.

The statement:

```
LGTH := FREAD(DFILE1,BUFFER,128);
```

reads an employee record from the file specified by DFILE1 into the array BUFFER in the stack. The statement:

```
FWRITE (LIST,BUFFER,-20,%320);
```

then displays this record on the terminal; $STDLIST has been opened with the FOPEN intrinsic and the resulting file number has been assigned to LIST. The statement:

```
DUMMY := FREAD(IN,BUFFER(30),5);
```

reads an employee number, entered on the terminal ($STDIN has been opened with the FOPEN intrinsic and the resulting file number has been assigned to IN), into the array BUFFER starting at word '0'. The statement:

```
FUPDATE(DFILE1,BUFFER,128);
```

then calls the FUPDATE intrinsic to update the last record accessed in the file specified by DFILE1. The contents of BUFFER (including the employee number entered from the terminal) are written into this record. Up to 128 words are written.

If the FUPDATE request was granted, a CCE condition code results.

The statement:

```
IF <> THEN FILERROR(DFILE,9);
```

checks for a "not equal" condition code and, if such is the case, calls the error-check procedure FILERROR. The procedure FILERROR prints a FILE INFORMATION DISPLAY on the terminal, enabling you to determine the error number returned by FUPDATE, then aborts the program's process.

Table 6-1 summarizes the characteristics of the intrinsics used in data transfer.

```
PAGE 0001 HEWLETT-PACKARD 32100A.05.1 SPL/3000 TUE, OCT 7, 10:32 AM
  00001000 00000 0 $CONTROL USLINIT
  00002000 00000 0 BEGIN
  00003000 00000 1 BYTE ARRAY DATAI(0:7):="DATAONE ";
  00004000 00005 1 ARRAY BUFFER(0:127);
  00005000 00005 1 INTEGER DFILE1,LGTH,DUMMY,IN,LIST;
  00006000 00005 1
  00007000 00005 1 INTRINSIC FOPEN,FREAD,FUPDATE,FLOCK,FUNLOCK,FCLOSE,
  00008000 00005 1 PRINT'FILE'INFO,QUIT,FWRITE,FREAD;
  00009000 00005 1
  00010000 00005 1 PROCEDURE FILERROR(FILENO,QUITNO);
  00011000 00000 1 VALUE QUITNO;
  00012000 00000 1 INTEGER FILENO,QUITNO;
  00013000 00000 1 BEGIN
  00014000 00000 2 PRINT'FILE'INFO(FILENO);
  00015000 00002 2 QUIT(QUITNO);
  00016000 00004 2 END;
  00017000 00000 1
  00018000 00000 1 <<END OF DECLARATIONS>>
  00019000 00000 1
  00020000 00000 1 DFILE1:=FOPEN(DATA1,%5,%345,128); <<OLD DISC FILE>>
  00021000 00011 1 IF < THEN FILERROR(DFILE1,1); <<CHECK FOR ERROR>>
  00022000 00015 1
  00023000 00015 1 IN:=FOPEN(,%244); <<$STDIN>>
  00024000 00024 1 IF < THEN FILERROR(IN,2); <<CHECK FOR ERROR>>
  00025000 00030 1
  00026000 00030 1 LIST:=FOPEN(,%614,%1); <<SSTDLIST>>
  00027000 00030 1 IF < THEN FILERROR(LIST,3); <<CHECK FOR ERROR>>
  00028000 00044 1
  00029000 00044 1 UPDATE'LOOP:
  00030000 00044 1 FLOCK(DFILE1,1); <<LOCK FILE/SUSPEND>>
  00031000 00047 1 IF < THEN FILERROR(DFILE1,4); <<CHECK FOR ERROR>>
  00032000 00053 1
  00033000 00053 1 LGTH:=FREAD(DFILE1,BUFFER,128); <<GET EMPLOYEE REC>>
  00034000 00061 1 IF < THEN FILERROR(DFILE1,5); <<CHECK FOR ERROR>>
  00035000 00065 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
  00036000 00070 1
  00037000 00070 1 FWRITE(LIST,BUFFER,-20,%320); <<EMPLOYEE NAME>>
  00038000 00075 1 IF <> THEN FILERROR(LIST,6); <<CHECK FOR ERROR>>
  00039000 00101 1
  00040000 00101 1 DUMMY:=FREAD(IN,BUFFER(3) ,5) ; <<EMPLOYEE NUMBER>>
```

Figure 6-4. FUPDATE Example (1 of 2)

```
00041000 00110 1 IF < THEN FILERROR(IN,7); <<CHECK FOR ERROR>>
00042000 00114 1 IF > THEN GO END'OF'FILE;
00043000 00115 1
00044000 00115 1 FUPDATE(DFILE1,BUFFER,128); <<EMPLOYEE RECORD>>
00045000 00121 1 IF <> THEN FILERROR(DFILE1,8); <<CHECK FOR ERROR>>
00046000 00125 1
00047000 00125 1 FUNLOCK(DFILE1); <<ALLOW OTHER ACCESS>>
00048000 00127 1 IF <> THEN FILERROR(DFILE1,9); <<CHECK FOR ERROR>>
00049000 00133 1
00050000 00133 1 GO UPDATE'LOOP; <<CONTINUE UPDATE>>
00051000 00140 1
00052000 00140 1 END'OF'FILE:
00053000 00140 1 FUNLOCK(DFILE1); <<ALLOW OTHER ACCESS>>
00054000 00142 1 IF <> THEN FILERROR(DFILE1,10); <<CHECK FOR ERROR>>
00055000 00146 1
00056000 00146 1 FCLOSE(DFILE1,0,0); <<DISP-NO CHANGE>>
00057000 00151 1 IF < THEN FILERROR(DFILE1,11); <<CHECK FOR ERROR>>
00058000 00155 1 END.
   PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00204
   NO. ERRORS=000; NO. WARNINGS=000
   PROCESSOR TIME=0:00:03: ELAPSED TIME=0:00:17
```

Figure 6-4.FREADDIR and FREADSEEK Example (2 of 2)

Table 6-1. Intrinsics for Data Transfer

Intrinsic	Data Transfer Uses
FREAD	Used for sequential read. May be used with fixed, variable, or undefined-length record files. 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, variable, 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. Use only with fixed or undefined-length record files. File must be opened with read, read/write, or update access. Successful read returns a 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. Use only with buffered fixed and undefined-length record files. File must be opened with read, read/write, or update access. Successful read returns a 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 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). Use only with fixed or undefined-length record files. File must be opened with update access. No multi-record update allowed. Successful date returns a CCE condition code; file error results in CCL condition code; end-of-file results in CCG condition code.

Relative I/O

In addition to the conventional random and serial access, MPE offers Relative I/O access (RIO). RIO is intended for use primarily by COBOL II programs; however, you can access these files by programs written in any language.

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 may be accessed in two ways: RIO access and non-RIO access. RIO access ignores the inactive records when the file is read serially using the FREAD intrinsic, and these records will be transparent. 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.

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 a specified number of records.
- Pointing: Set the record pointer to a specified record.
- 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.

The FSPACE intrinsic may be used only with files that contain fixed-length or undefined-length records. Variable-length record files are not allowed. FSPACE may not be used when you have opened your file with append access and the file system will return a CCL condition if you attempt to use it in that case. Attempted spacing beyond the end-of-file results in a CCG condition, and the pointer will not be changed.

Pointing

To request a specific location for the record pointer to indicate, use the FPOINT intrinsic. The 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).

The FPOINT intrinsic may be used only with files that contain fixed-length or undefined-length records. Variable-length record files are not allowed. FPOINT may not be used when you have opened your file with append access, and the file system will return a CCL condition if you attempt to use it in that case. Attempting to point beyond the end-of-file results in a CCG condition, and the pointer will not be changed.

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. The FCONTROL syntax in this case would be:

FCONTROL (*filenum,5,dummy'param*);

Issuing this intrinsic call will set the record pointer to record 0. You may use FCONTROL with fixed-length, variable-length, or undefined-length record files, and you may use it with any access mode.

Note

FCONTROL5 has a special meaning when used with append access. The file system will set 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 will be set to the end of the file so no data will be overwritten.

For more information about the FSPACE, FPOINT, and FCONTROL intrinsics, consult the *MPE V Intrinsics Reference Manual* (32033-90007).

Transferring Files

MPE provides facilities for transferring files between groups, accounts and different systems.

Inter-Group Transfers

To transfer a file from one group to another within the same account, use the RENAME command, simply naming the new group in the second parameter. For example:

```
:RENAME MYFILE.GROUP1,MYFILE.GROUP2
```

In this example, GROUP1 is the old group and GROUP2 is the new group.

Note

> To use RENAME in this way, you must be the creator of the file and have SAVE access to the group named in the second parameter (GROUP2 in the previous example). In addition, both groups must be in the system domain or must both reside on the same private volume set (renaming of files across volume sets is not allowed).

Inter-Account Transfers

To transfer a file from one account to another, proceed as follows:

1. Logon to the computer under the account presently containing the file.
2. Enter the RELEASE command to temporarily suspend any file system security provisions covering the file. For example:

   ```
   RELEASE FILEX File name
   ```

   You can enter this command only if you are the creator of the file.
3. Log off from this account and logon under the account to which the file is to be transferred.
4. Run the File Copier Subsystem (FCOPY) to copy the file from the old account into this account. For example:

   ```
       :RUN FCOPY.PUB.SYS
       >FROM=FILEX.GROUPA.ACCT1;NEW;TO=FILEX.GROUPA.ACCT2
   ```

   In this example ACCT1 is the old account name and ACCT2 is the new account name (optional entry).

Note

> The renaming of files across volume sets is not allowed, since this would require that the operation physically transfer the file between different volume sets.

   A copy of FILEX now exists under GROUPA of ACCT2; the original FILEX still exists under GROUPA of ACCT1.

5. Log off from the present account and logon again under the account containing the original file.
6. Restore the security provisions to the original file by entering the SECURE command:

```
:SECURE FILEX
```

Or, if you want only one copy of the file in the system, delete the original file by entering the PURGE command:

```
:PURGE FILEX
```

Note

To use the above commands, you must be the creator of the file. Steps 1 through 3, and 5 through 6, can be avoided if the file security for ACCT1 (the old account) allows read access from other accounts.

Inter-System Transfers

You can transfer one or more files between systems by copying them from their present system onto magnetic tape or serial disk in a special format, transporting that tape or disk pack to the new system, and loading the tape contents into the new system. To permit you to do this, however, the accounts, groups, and users to which the files belonged on the old system must also be defined on the new system. The technique for accomplishing this transfer involves the STORE command (to write the files to tape) and the RESTORE command (to copy the files from the tape into the new system).

For example, to store FILEZ on tape for transporting to another system, enter:

```
FILE TP;DEV=TAPE
STORE FILEZ;TP <------------------ Back references tape
```

Mount a tape and allow the file to be stored on it. Take the tape file to the new system, mount the tape, and copy the file into the system by using the RESTORE command:

```
FILE TP;DEV=TAPE
RESTORE TP; FILEZ
```

You can also transfer files by copying them to magnetic tape or serial disk via FCOPY and transporting that tape or disk pack to the new system and loading it. The method for doing this is discussed in the *FCOPY Reference Manual* (03000-90064).

Buffered Input/Output

A buffer is an area maintained by the file system outside of a user's stack. It serves as an intermediate area for data transfer. The file system can move data from a file to a buffer, and from there to your stack, or it can move the data from your stack to a buffer and from there to a file.

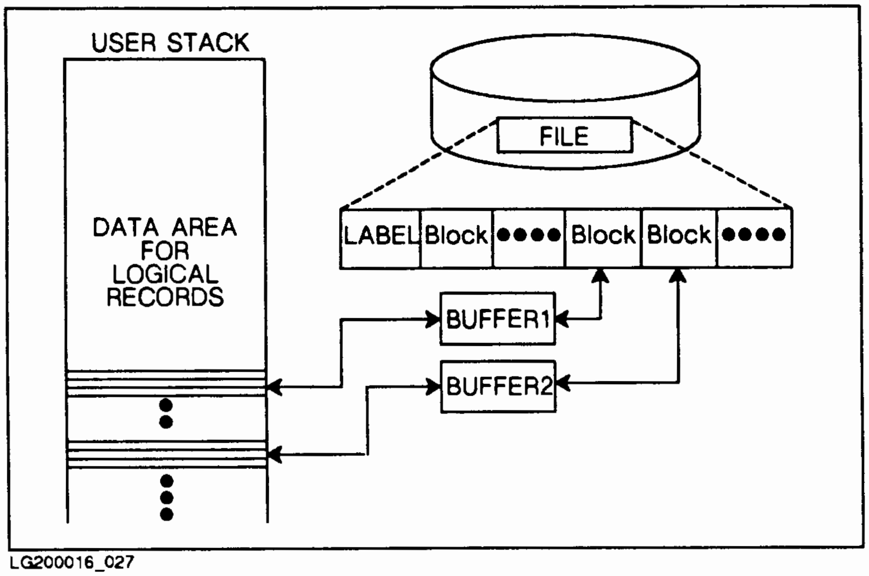


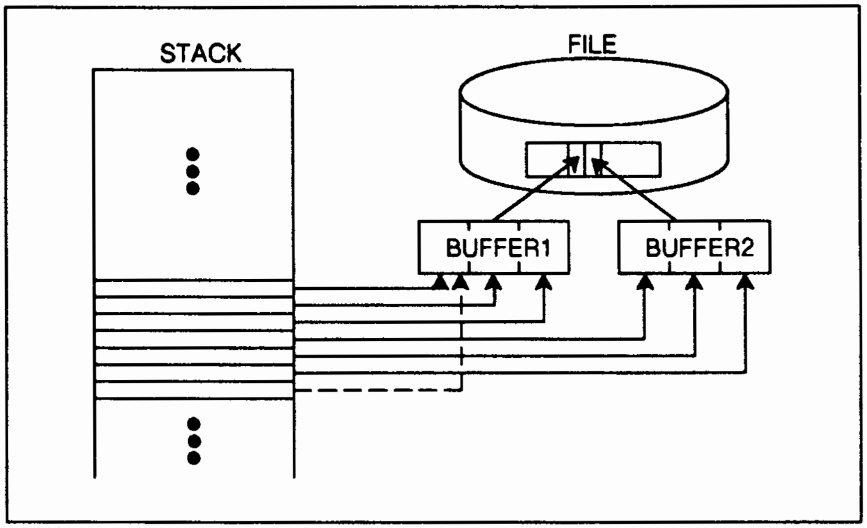
Figure 6-5. Data Transfers Using Buffers

Your buffers will be the same size as the blocks for your file. Every read or write of data between the file and a buffer will move one ˈblock of data from or to the buffer. Data is moved between the buffer and your stack in units of one logical record each.

So, if your program is reading data from a file into your stack, it will move a block of data from the file into a buffer, and then move the data from the buffer to your stack one logical record at a time. When it has moved the entire block, another block of data will be moved to the buffer and will be moved record by record to your stack. On the other hand, if your program is writing data from your stack to a file, it will write data to a buffer one logical record at a time. When the buffer is filled, it will contain a block of data. This block will be moved or posted to your file. When the buffer has been posted to the file, it is ready to receive more records from your stack.

Figure 6-6 shows the transfer of data using two buffers. The blocking factor of the file is three, so three logical records fit into each block. Each buffer is the size of one of the file's blocks.

A program is writing data from the user's stack to the file. The first three logical records are written to the first buffer. Now that it is filled, this buffer is posted to the file and the fourth logical record is written to the second buffer. When the fifth and sixth records have been written to this buffer, this buffer is also posted to the file, and the seventh logical record is written to the first buffer.



LG200016_028

Figure 6-6. Buffer Operation

You may specify the number of buffers you want to use with your file by issuing a `FILE` command or using the *numbuffers* parameter in the `FOPEN` intrinsic. If you do not specify the number of buffers, the file system will assign the default of two buffers. You may have one or more buffers, to a maximum of sixteen.

Note Although you may specify a maximum of 16 buffers, any number beyond 3 does not usually increase input/output efficiency and occupies needless space in main memory. If you request 0 buffers, the file system will override this and supply the standard default of 2.

Note For files input or output at interactive terminals, you need not specify any buffer parameter. A system-managed buffering operation is always used for terminals. If you do specify any buffers for a terminal, the file system will override this specification and assign no buffers.

The maximum total buffer space for an individual file is 32,000 words. This means that if a file has one buffer, that buffer may be up to 32,000 words in size. If a file has two buffers, they may each be up to 16,000 words in size and so on. If the total buffer size you request is too large (that is, *blocksize* x *numbuffers* > 32,000) an error, "out of virtual memory", will result.

Note

For magnetic tape files, the maximum size of a data transfer is 8K words (8,192 words). The actual transfer size can be larger depending on the specific magnetic tape device.

Why Buffer Transfers?

There are two major reasons for buffering data transfers: buffering results in automatic blocking and deblocking of logical records and provides the ability to do anticipatory reading.

Automatic Blocking and Deblocking

Your program may try to locate a particular logical record. All data transfers, however, occur in units of blocks. With buffering, the file system will handle the details of locating the desired record in a particular block.

Anticipatory Reading

This technique effectively permits the overlapping of input/output requests, often significantly reducing the time required to process a file. Anticipatory reading involves moving data from a file into a buffer before it is needed, so it can be moved into the user's stack immediately when it is needed. For instance, if your program is reading data from a sequential disk file in blocks of four records each, upon the first read request, the file system automatically moves the first four records from the file to the first buffer (Buffer 1) and the next four records into the second buffer (Buffer 2). When your program has read all four records from Buffer 1 and accesses the first record in Buffer 2, the file system automatically moves the next four unread records in the file into Buffer 1 so that they will be immediately available for any upcoming read request. When your program reads all records in Buffer 2, the file system moves another four records into Buffer 2, continuing in this fashion until the program terminates access to the file. Anticipatory reading is most effective in purely sequential access operations, but it can also be used in conjunction with the `FREADSEEK` intrinsic when you wish to access records nonsequentially.

Unbuffered I/O

On occasion, you may wish to avoid the use of buffers altogether. This may be the case, for instance, when you are transferring records in large blocks. These can require excessive amounts of memory for the data transferred and additional overhead for pointers and file-access information required to maintain the buffers. Furthermore, certain file-access modes prohibit the use of buffers. For example, multi-record (MR) access and NOWAIT input/output, are incompatible with buffering. If you request buffering in such cases, the file system will override your request and allocate no buffers.

To expressly specify no buffering, enter the NOBUF keyword parameter in a FILE command, as follows:

```
FILE BIGDATS;REC=-4096,16,F;NOBUF<------------ Specifies no buffers
```

Note

During an unbuffered transfer, your stack will be frozen in memory because the I/O processor needs the absolute addresses of the records it processes. Also, your process will suspend execution during the transfer.

When you do not use buffering, you may transfer your data in blocks only. The file system will not deblock logical records for you. For this reason, it is more efficient to use unbuffered transfers when you are copying files.

NOWAIT Input/Output

Normally, when a program issues a request for input/output, control does not return to the program until that request has been satisfied. However, the file system allows programs to bypass this convention by initiating input/output requests with control returning to the program prior to the completion of the request. This feature is known as NOWAIT input/output.

You may specify NOWAIT input/output in your FOPEN call to your file, or you may request it in a FILE command that references the file:

```
      File name Requests no buffering
  |
  |
  |
  |
  V
FILE QUICKFL; NOBUF; NOWAIT
     ^
     |
     |
          Requests NOWAIT input/output
```

Note

To ultimately confirm input/output completion, your program must call the `IOWAIT` intrinsic after the request.

To use the NOWAIT feature, your program must be running in Privileged Mode. A NOWAIT request implies that no buffering is used.

The normal checks and limitations that apply to standard users in MPE are bypassed in Privileged Mode. It is possible for a Privileged Mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard will investigate and attempt to resolve problems resulting from the use of Privileged Mode code. This service, which is not provided under the standard service contract, is available on a time and materials billing basis. Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.

How Many Buffers?

How do you choose the number of buffers for your file? The implications of the number you choose are given in Table 6-2.

Table 6-2. Implications of Number of Buffers

0 (NOBUF)	User program suspends execution during every transfer. User's stack is frozen in memory during transfer. Can only transfer physical records.
1	User program suspends when necessary logical record is not in buffer. User's stack is not frozen in memory.
2	User program may not suspend; allows parallel processing. Buffer usage is alternated.
3 (or more)	User program may not suspend even under heavy I/O load. Useful for local set of frequently accessed records.

Note

Table 6-2 lists implications, not recommendations. The most efficient number of buffers will depend upon your particular application.

Multi-Record Mode

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 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 the multi-record (MR) mode in your FOPEN call or FILE command. For example:

```
FILE BIGCHUNK; REC=-256,1,U;NOBUF;MR <------ Specifies multi-record mode
```

The essential effect of multi-record mode is to make it possible to transfer more than one block in a single read or write. This mode effectively ignores block and sector boundaries, and will permit transfers of as much data as you wish (specified in the 'TCOUNT' parameter of the READ or WRITE request). It will not, however, break up blocks or sectors and every transfer must begin on block and sector boundaries. In order to take advantage of multi-record mode, you should specify the NOBUF option in your FILE command or FOPEN call.

When you read from a file in multi-record mode, you may not read beyond the end-of-file indicator. When you write to a file in multi-record 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 V Intrinsics Reference Manual* (30000-90010). The transfer length will be returned in the TLOG parameter of FCHECK.

For maximum efficiency in multi-record mode transfers you should build your file with a blocksize that is a multiple of 128 words (one sector). When you do this, data can be transferred from disk to your stack (or vice versa) with a single physical I/O request. When your file's blocksize is not a multiple of 128 words, a separate physical I/O request will be issued by MPE to transfer each block.

Buffer Control Intrinsics

Certain intrinsics permit you to exert a degree of control over the way the file system manages your buffers. The FSETMODE and FCONTROL intrinsics can be used in this way.

If you issue a call to FSETMODE with a *modeflag* value of 2, you set the Critical Output Verification bit. When you do this, every time a full buffer is posted to your file, your process suspends execution while the transfer is made, and remains suspended until the posted buffer is verified as complete. Use FSETMODE in this way only with buffered files; it is ineffective and unused in the NOBUF case.

When you issue an FCONTROL intrinsic call with a control code of 2, you are requesting that the file system "complete I/O". This will force the posting of all buffers that have been changed since the last time they were posted, and will mark the buffers as empty. Your process will suspend execution until these operations are complete. Use FCONTROL in this way only with buffered files. It is ineffective in the NOBUF case.

The FCONTROL intrinsic can also be used with a control code of 6, to specify "write EOF". When you issue this call for a buffered file, the file system will post all buffers that have been changed since their last posting and your process will suspend execution until posting is complete. The buffers will then be marked empty. For both buffered and unbuffered files, issuing FCONTROL 6 will update the end-of-file indicator and the extent map in the file label; updating the extent map will protect newly allocated extents from being lost in the case of a system crash.

The FSETMODE and FCONTROL intrinsics are discussed in detail in the MPE V Intrinsics Reference Manual (32033-90007).

Section Divider

7. File Security

7

File Security

There are two types of file security offered on the HP 3000 Computer System, they are:

- The Standard File System Security Provision
- The Access Control Definition (ACD)

Standard File System Security Provision

Associated with each account, group, and individual file is a set of security provisions that specifies any restrictions on access to the files in that account, group, or particular file.

Note

These provisions apply to disk files only.

These restrictions are based on three factors:

- Access Mode
- Type of User
- Use of Private Volumes

The security provisions for any file describe what modes of access are permitted and to which users.

Specifying and Restricting File Access by Access Mode

When a program opens or creates a file, it can define the way it can access the file by specifying a particular access mode for the file. These specifications apply to files on any device. In addition, for files on disk, a program can also restrict acces

The access types that can be specified by a program are listed in Table 7-1.

Table 7-1. 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, line printer, as well as tape, disk, and terminal output files.
Write (Save) Only	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 will not be deleted, but a normal write will replace 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.
Input/Output	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.

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 you select.

Because they are best explained by example, the effects of each access mode upon these factors are summarized in Table 7-2. This file contains ten 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

Table 7-2. 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) Only	10	0	10
Append	10	10	10
Input/Output	10	0	10
Update	10	0	10

Suppose 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 delete existing records. You can override the default specifications by using the FILE command to request append access to the file, as follows:

```
FILE TASK; DEV=TAPE; ACC=APPEND {requests append access}
RUN PROGN
```

Suppose 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 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; ACC=UPDATE {requests update access}
RUN PROGN
```

Consider a program that reads input from a terminal (file name INDEV) and directs output to a line printer (OUTDEV). You can redirect the output so that it is instead 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 security system, the mnemonic codes used to reference them, and their complete definitions are listed in Table 7-3.

Table 7-3. User Type Definitions

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.
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 log on 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 filename 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 with system manager capability..

At the account level, five access modes are recognized:

```
R = Read
A = Append
W = Write
L = Lock
X = Execute
```

Also at the account level, two user types are recognized:

```
ANY = Any User
AC = Account Member
```

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.
- For all other accounts, the Read, Append, Write, Lock, and Execute access modes are limited to account members.

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 provis 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 manager for that group.

At the group level, six access modes are recognized:

```
R = Read
A = Append
W = Write
L = Lock
X = Execute
S = Save
```

Also at the group level, five user types are recognized:

```
ANY = Any User
AC = Account Member
GL = Group Librarian
GU = Group User
AC = Account Member
```

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; Append, Write, Save, and Lock access are limited to Account Librarian group user.
- For all other groups in the account, Read, Append, Write, Save, Lock, and executing access are limited to group users.

File Level Security

When a file is created, the security provisions that apply to it are the default provisions assigned by MPE 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, the creator of the file (and only this individual) can change the file level security provisions. Therefore, 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

Note

If no security provisions are explicitly specified by the user, the following provisions are assigned at the file level by default.

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 group level allow access to group users only. 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 7-4. 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 log on group and home group.
- Read and Execute 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 7-4. Default Security Provisions

FILE REFERENCE	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 the account and group levels are managed only by users with the system manager or account manager capabilities respectively, but you can change the security provisions for any disk file 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 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 log on 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 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 the command:

```
ALTSEC FILEZ;(X:GU;R,W:GL)
```

You 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. For more information, refer to the ALTSEC command in the *MPE V Commands Reference Manual* (32033-90006).

Suspending and Restoring Security Provisions

You may temporarily suspend the security restrictions on any disk file 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 does not modify the file security settings recorded in the system. It merely bypasses them temporarily. The RELEASE command remains in effect until you enter the SECURE command in this or a later job/session.

To release the security provisions for the file named FILESEC in your log on group, enter:

```
RELEASE FILESEC
```

If the file has a lockword and you wish to remove that as well as all account, group, 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.

Access Control Definitions (ACDs)

Access Control Definitions (ACDs) contain a list of the users and the access mode each user or group of users has to the file. When an ACD is initiated, all previous Standard File System Security Provisions and lockwords are ignored. Only the ACD in eff

- *Modes* can be a combination of any of the following types of access/permissions and can be established for any user, account, and group level:

```
R = Read access
W = Write access
L = Lock access
A = Append access
X = Execute access
None = No access
RACD = Copy or Read permission (to the ACD only)
```

- *User specifications* are defined as a fully qualified user name (for example, JOHN.DOE). Therefore, an ACD can be defined as a combination of *modes:user specifications,* for example:

```
ACD=(R:JOHN.DOE;W,A,L:@.DOE,@.PAYROLL;R:@.@)
```

Who is the ACD Owner

The owner of an ACD associated with a file is:

- The creator of the file the ACD is associated with
- A user with AM capability for the account where the file resides
- A user with SM capability

Note

The owner of an ACD is the only user capable of adding, changing, or deleting an ACD.

How are ACDs Used

ACDs are used to determine if a user trying to access a file is authorized. When an ACD is used to secure a file, it will be the only mechanism used to determine access rights. The Standard File System Security Provision will not be used.

To access a file, the system will execute the following checks at `FOPEN`.

1. Is the user one of the following:
   - System Manager (SM capability)
   - Account manager (AM capability)
   - Creator of the file

   If you are one of these three, you can access the file. If you are not one of these three, the next check is made by the system.
2. Is there an ACD associated with the file? If yes, the system evaluates the user against any user specifications given in the ACD:
   - The user name is compared to all specific names in the ACD.
   - The user name is compared to all wild cards that include the account name in the ACD.
   - The user name is compared to any wild cards used to represent the system.
   - If there is no match then, the user is not granted access.
3. If there is no ACD assigned, the system will default to the Standard File System Security Provision to determine if the user is granted access.

How to Create an ACD

An ACD can be created and associated with a file in one of two ways:

- Create a file (*^acdfilename*) which contains the ACD *modes:user specifications* only, and then assign it to the file.
- Assign the ACD, *modes:user specifications (pair_spec)* explicitly to the file.

To create an ACD, issue the ALTSEC command as follows:

```
ALTSEC filename[,filetype];NEWACD={^acdfilename}
       {(pair_spec)}
```

allows the file FILEX to be read by all groups on all accounts, allows Write access by all groups on account DOE, and allows Execute access by JOHN.DOE only. Or, the command could be written as:

```
ALTSEC FILEX,JOHN.DOE;NEWACD=^ACDFILE
```

where ^ACDFILE is the *^acdfilename* containing the *modes:user specifications* only, as found in the previous example. Refer to the ALTSEC command in the *MPE V Commands Reference Manual* (32033-90006) for more information.

Note

Wild cards are not allowed for file names.

How to Read, List and Copy an ACD

The ALTSEC command can be used to copy an ACD:

```
ALTSEC filename[,filetype];[COPYACD={sourcefilename}[,sourcefiletype]]
```

To copy an ACD from one file to another, the user must have Read ACD access (RACD) to the source file or be the owner of the source and destination files. Reading an ACD is performed when copying or listing an ACD.

How to Modify an ACD

The ALTSEC command can be used to modify an existing ACD. There are three types of modifications that can be made with the ALTSEC command.

- Add
- Replace
- Delete

To add the *modes:userspecification* pairs to an ACD that does not contain one, enter:

```
ALTSEC filename[filetype];ADDPAIR= {(pair_spec)}
        {^acdfilename}
```

The modifications can be given explicitly with (*pair_spec*) or can be the contents of the *acdfilename* file.

To replace the user specifications given in the (*pair_spec*) or *acdfilename*, enter:

```
ALTSEC filename[,filetype];REPPAIR= {(pair_spec)}
        {^acdfilename}
```

The modifications can be given explicitly with (*pair_spec*) or can be the contents of the *acdfilename* file.

To delete a *userspecification* enter:

```
ALTSEC filename[,filetype];DELPAIR= {(userspecification)}
        {^acdfilename}
```

This will delete those pairs associated with the *userspecification* or the *acdfilename*.

How to Delete an ACD

The ALTSEC command can be used to delete an ACD:

```
ALTSEC filename[,filetype];DELETE
```

The owner of an ACD is the only one allowed to delete an ACD.

Section Divider

8. Interprocess Communication

8

Interprocess Communication

Interprocess Communication

Interprocess communication (IPC) is a facility of the file system which permits multiple user processes to communicate with one another in an easy and efficiently. To accomplish this, IPC uses message files as the interface between user processes. These 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.

Occasionally a process may attempt to read a record from an empty message file, or write a record to a message file that is full. In such cases, the file system will usually cause the process to wait until its request can be serviced. That is, until another process either writes a record to the empty file or reads enough records to take a block from the full file.

There is a unidirectional flow of information between a given process and a message file. A process opening the file with read access, identified as a reader, may only read from the file, and not write. A process opening the file with write access, identified as a writer, may only write to the file and not read. If it is necessary for the same process to read and write, it may open the file twice, once as a reader and once as a writer. More than one message file may be associated with a process, and the process may be configured as a reader to some of the files and as a writer to others. A given message file typically has one reader, though more are allowed, and one or more writers.

Applications for IPC exist wherever it is necessary for processes to communicate with one another. In the case of a father process with several sons, message files may serve as interfaces between the processes. Through one file, the father may direct the activities of the sons; through another, the sons may inform the father of their progress. Message files may also aid object managers during data base operations. Several writers may send information to a file which serves as the single source from which the data base process actually receives the information.

Operation

Message files are maintained and manipulated by several intrinsics. The FOPEN, FREAD, FWRITE, FCONTROL, and FCLOSE intrinsics operate upon the files to yield a unidirectional, first-in-first-out message queue:

FOPEN — Establishes a connection to a message file. With FOPEN, a user process identifies itself as either a reader or a writer. Readers access the front of the message file and writers access the end of the message file. Incompatible parameters that are specified with FOPEN are adjusted. For example, since messages are read or written to the file one record at a time, a multi-record parameter is corrected. If FOPEN is used to access a new file, a new message file is created.

Note

The Access Type (bits (12:4) of the Access Options) specifications are interpreted slightly different than for writers of conventional files. In one case, if a writer is the first accessor to a message file, the file's contents are purged; in another case, the writer simply appends records to the tail of the file.

FREAD — Reads one record from the start of a message file. The record is copied to the reader's TARGET area and is logically deleted from the message queue. The next record is now at the beginning of the file. If a process tries to read from an empty message file which writers are accessing, the file system waits until a writer process enters a record to the file. If there are no writers associated with the message file, an end-of-file indication, CCG, is returned.

Note

If the message file is empty and there no writers, the reader process will wait if there is an FCONTROL 45 in effect, or if this is the first FREAD after the reader's FOPEN.

FWRITE — Appends one record to the end of a message file. If a process tries to write to a full message file which readers are accessing, the file system waits until a reader process has read a block of records from the file. If there are no readers associated with the message file, an end-of-file indication, CCG, is returned.

Note

If the message file is full and there are no readers, the process will wait if there is an FCONTROL 45 in effect, or if this is the first FWRITE after the writer's FOPEN.

`FCONTROL` Supplies various control functions to a process that is using a message file. These control functions permit a process to take advantage of the additional features of IPC, which are discussed in detail later in this chapter.

`FCLOSE` Breaks a process' connection with a message file. If the process reopens the same file later, it may do so as either a reader or a writer, regard less of what it was previously.

Additional Features

Besides the regular attributes of IPC and message files, other features are available for use with these facilities. Writer IDs, nondestructive reads and software interrupts are specifically intended for use with IPC. Copy access is a general enhancement to the file system. `FREADS` and `FWRITES` to message files can use time-outs.

Writer IDs

When a writer process opens a message file, the file system assigns a unique 16-bit ID number to the writer. Each record the process writes to the message file is prefixed with this number by `FWRITE`. When the writer closes the file, the ID number is no longer associated with the process and may be reused. A writer posts his OPEN record when the first write to the file takes place. The `CLOSE` record is written if any records have been written while the file was open. This was designed into IPC to eliminate the need for `FOPEN` to wait for file space. It also allows a writer to open/close the file without having any effect on it. Record prefixes and open/close records are usually transparent to the readers of the message file, but by issuing an `FCONTROL 46`, the reader process may see them. The interested reader may use the writer IDs to determine the source of the records it is receiving.

Time-outs

A reader or a writer process may limit the length of time it will wait to be serviced. By issuing an `FCONTROL 4`, a reader may specify the maximum number of seconds it will permit the file system to keep it waiting for a record to be written to an empty message file. A writer may also use `FCONTROL 4` to specify the maximum number of seconds it will wait for a block of records to be read from a full file.

Copy Access

When records are read from a message file, FREAD logically deletes them as it reads. In order to copy a message file without destroying it, the file must be opened with the file copy option specified in the aoptions of the FOPEN, or the COPY keyword must be specified in a FILE command. When this option is selected, the message file is treated as a standard sequential file rather than as a message queue and may be copied safely. The file may then be read by logical record or by block and information may be written to it by block.

Note

In order to access a message file in copy mode, a process must have exclusive access to the file.

Nondestructive Read

By issuing an FCONTROL 47, a reader may avoid deleting the next record it reads. The record will remain at the head of the message queue. This feature differs from the copy access feature in that it is a temporary condition. The second FREAD following the FCONTROL 47 will reread the record and delete it in the usual manner.

Software Interrupts

You may specify that your FREAD and FWRITE completion processing be done with an "interrupt" procedure supplied in your program. An FREAD or FWRITE intrinsic call is required to start the I/O request. As with NOWAIT I/O, the FREAD/FWRITE intrinsics return control to your program immediately after the request is initiated. When the request completes, your program is "trapped" to your interrupt procedure to process the I/O completion.

Using IPC

Message files can be created in several ways. When a user process opens a new file and indicates in the foptions that it will be a message file, the FOPEN intrinsic creates the new message file. In order to create a message file with the BUILD command, use the MSG keyword. For example, to build a message file named SARA, enter:

```
BUILD SARA; MSG
```

A new message file may also be defined with a FILE command. Use the MSG keyword for a new file:

```
FILE LISBETH, NEW; MSG
```

A message file named LISBETH is indicated.

When you perform a LISTF,2 command, message files will be identified by an "M" in the third column of the TYP field; SARA is identified here:

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

                 SIZE     TYP     EOF     LIMIT     R/B  SECTORS    #X MX


SARA            128W     VBM     0       1031      1        258       1 8
```

Other types of files are similarly indicated by a token in the TYP field:

R—identifies a Relative I/O file

O—identifies a Circular file

A blank in the third column indicates a standard MPE file.

Occasionally, you might create a message file and specify a certain number of records for the file to contain, only to discover that the file system has allocated more records than you requested. The reason for this is that the file system leaves room to maintain the necessary internal structure for the message file. The file system has four basic rules for establishing this structure when the message file is created:

1. The file system adds two records to the requested number to allow for a minimum of one open and one close operation.
2. The requested number of records is rounded up to fill an even number of blocks.
3. The file system adds an extra block to the message file for the file label to occupy. (This block is transparent.)
4. The file system assigns the same number of blocks to each extent.

For example, suppose you want to create a message file named ODDSIZE:

```
BUILD ODDSIZE; MSG; REC=,3; DISC-51,8
```

You have specified a message file with fifty-one records, three records per block, that occupies eight extents. The file system will adjust the number of records to conform to the rules for message file structure:

1. The file system adds two records to allow for one open and one close indication; the number of records goes from 51 to 53.
2. The number of records is rounded up to 54 to provide an even number of blocks. With three records per block, 54 records will fill 18 blocks.
3. An additional block is added to the file to accommodate the file label. Now the file contains 19 blocks.

4. The eight extents must all be the same size, so the number of blocks is increased from 19 to 24. Each extent now contains three blocks.

Of the 24 blocks in ODDSIZE, 23 are data blocks and one contains the file label, which is invisible to you. With three records per block, 23 blocks contain a total of 69 data records.

Note

In addition to adjusting the number of blocks in a message file, the file system adds a certain amount of space to each block for *overhead,* six bytes will be added to each record, and four bytes will be added for each block.

Features of Intrinsics for Message Files

There are a few features of several intrinsics which apply specifically to message files. Most of these features are found in FOPEN and FCONTROL, but several other intrinsics are also affected.

Some of the parameters of the following intrinsics contain more than one piece of information within each 16-bit word. When this is the case, data fields are described in the following format: (*n*:*m*), where *n* is the first bit of the field and *m* is the number of consecutive bits in the field. For example, the *FOPTION* field for File Type, described below, occupies bits (2:3), or bits 2, 3 and 4.

Parameters not mentioned in the following descriptions retain their normal range of values and their normal default values.

FOPEN

FOPTIONs: (2:3) - File type. Determines the type of file to create for a new file. If the file is old, this field is ignored.

000 -	Ordinary file
001 -	KSAM file
010 -	Relative I/O file
100 -	Circular file; discussed later
110 -	Message file

Note

The Default Designator *FOPTION,* bits 10 through 12, offers several choices for default file designators. Any value used other than 0 for *filename* will override the File Type field.

(8:2) - Record format. Message files are always internally formatted as variable-length record files. However, a message file can appear as a fixed file to an opener. There is no difference for a writer, but a reader will have the portion of his target area which exceeds the record filled with blanks (for an ASCII file) or zeros (for a binary file).

00 -	Fixed
01 -	Variable
10 -	Undefined; changed to variable

AOPTIONs: (3:1) - File copy. This feature permits a message file to be treated as a standard sequential file, so it can be copied by logical record or physical block to another file.

0 - The file will be accessed in its native mode. A message file will be treated as a message file.

1 - The file is to be treated as a standard, sequential file with variable-length records. This allows nondestructive reading of an old message file at either the logical record or physical block level. Only block level access is permitted if the file is opened with write access. These blocks are checked for proper message file format to prevent incorrectly formatted data from being written to the message file while it is unprotected.

Note

In order to access a message file in copy mode, a process must have exclusive access to the file.

Setting this bit on causes all the remaining file parameters to have their normal defaults.

(5:2) Multi-access mode. This feature permits processes located in different jobs or sessions to open the same file.

00 - No multi-access. The file system changes this value to 2 to allow global multi-access.

01 - Only intra-job multi-access allowed. This is the same as specifying the MULTI option in a `FILE` command.

10 - Inter-job multi-access allowed. This is the same as specifying the GMULTI option in a `FILE` command.

11 - Undefined. If this is specified, the `FOPEN` will be rejected with an error code of 40: `ACCESS VIOLATION`.

(7:1) - Inhibit buffering. For message files, the file system sets this bit off.

0 - Read by logical record

1 - Read by physical block

Writers must open message files with `NOBUF` if they are in copy mode. Access of the file is block by block.

Note

Readers may open a message file with `NOBUF` if they are in copy mode. This determines whether they will be accessing the file record by record or block by block.

(8:2) - Exclusive. The values for this field are the same as for any disk file, but they have different meanings for the readers and writers of a message file:

User Value	**Meaning**
EXCLUSIVE	One reader, one writer.
SEMI	One reader, multiple writers.
SHARE	Multiple readers and writers
Default	One reader, one writer.

(11:1) - Multi-record. For message files, the file system sets this bit to 0, except in copy mode.

(12:4) - Access type. These bits specify whether the user will be a reader or a writer process.

0000 READ access only. The **FWRITE** intrinsic cannot reference this file. This access type requires both read and write access capability to the file. A process that has opened a file with this access type is a *reader.*

0001 WRITE access only. If this is the first accessor to the file and the process has write access capability, then the file's contents are purged. If this is not the first accessor to the file, the file system sets this access type to APPEND. The **FREAD** intrinsic cannot reference this file. A process that has opened a file with this access type is a *writer.*

0010 WRITE SAVE access. The file system sets this to APPEND access.

0011 APPEND access only. The **FREAD** intrinsic cannot reference this file. This access type requires append capability to the file. A process that has opened a file with this access type is a *writer.*

device This field is relevant only if this is a new file. The DEVICE field must either be omitted or specify a disk; specification of any device other than a disk opens the device. When this occurs, the file is no longer a message file.

numbuffers (0:11) - Ignored.

(11:5) - Value between 2 and 31; default is 2. This parameter must not exceed the physical record capacity of the file.

filesize The number of records is rounded up to completely fill the last block and to make the last extent the same size as the other extents. Two additional records are included for the open and close records.

FCONTROL

The control codes that deal specifically with IPC are shown in Table 8-1. Those not mentioned here are invalid when IPC is being used.

Table 8-1. IPC Control Codes

CONTROL CODE	PARAM	DESCRIPTION
2	-	Complete all I/O; ignored in the case of message files.
3	-	Read hardware status word.
4	integer	Set time-out interval. This applies to both FREADs and FWRITES. The timeout will be armed at the beginning of the I/O request and cleared when the I/O completes. PARAM specifies the length of the time-out in seconds. A value of zero disables timeouts in the file.
6	-	Write end-of-file, used to verify the state of the file by writing out the file label and buffer area to disk. This ensures that the message file can survive system crashes. No eof is written.
43	-	Abort NOWAIT I/O. A CCG condition code is returned if an outstanding I/O operation has completed. An IOWAIT must be issued to finish the request.
45	TRUE	Enable extended wait. Permits a reader to wait on an empty file that is not currently opened by any writer, or a writer to wait on a full file that has no reader. This FCONTROL Will remain in effect until FCONTROL 45 is issued with a PARAM value of FALSE.
	FALSE	Disable extended wait. Specifies that when an FREAD encounters an empty file that has no writer, or an FWRITE encounters a full file that has no reader, it will return an end-of-file condition. (Default.)

Table 8-1. IPC Control Codes (continued)

CONTROL CODE	PARAM	DESCRIPTION
46	TRUE	Enable reading the writer's ID. Each record read will have a two-word header. The first word will indicate the type of record: 0 - date record 1 - open record 2 - close record The second word will contain the writer's ID number. If the record is a data record, the data will follow the header; open and close records contain no more information.
	FALSE	Disable reading the writer's ID. Only data is read to the reader's TARGET area. The open and close records are skipped and deleted by the file system when they come to the head of the message queue, and the two-word header is transparent to the reader. (Default.)
47	TRUE	Nondestructive read. The next `FREAD` by this reader will not delete the record. Subsequent `FREAD`s will be unaffected.
47	FALSE	The next `FREAD` by this reader will delete the record. (Default.)
48	PLABEL	Arm soft interrupts. PARAM contains the external-type label (*plabel*) of your interrupt procedure. In SPL it is passed as a parameter by placing an "at" sign (@) before the procedure name. Also if *aoptions* (4:1) was set to 0, option 48 resets it to 1. Be sure to use `IOWAIT` or `IODONTWAIT` if you use control code 48. If the value of PARAM is O, the interrupt mechanism is disabled for this file.

FCHECK There is one error message that is returned only when using IPC:

```
151 CURRENT RECORD WAS LAST RECORD WRITTEN BEFORE SYSTEM CRASHED
```

This message is returned when this record is read following system startup.

FGETINFO

The value returned in RECSIZE will indicate the user's data record size, and the value returned in EOF will indicate the number of data records, unless an FCONTROL 46 is in effect. When an FCONTROL 46 is in effect, the value returned in RECSIZE will be the size of the user's data records, including the two word header. The number of records returned in EOF will include open, close and data records.

The value returned in BLKSIZE reflects the actual blocksize of the file. When the file is created, the blocksize is computed by the following algorithm:

```
BLOCKSIZE:=((RECORDSIZE+3)*BLOCKING FACTOR)+2
```

where RECORDSIZE and BLOCKSIZE are in words. For example, with a recordsize of 100 words and a blocking factor of 10, the blocksize would be 1032 words.

FFILEINFO

Three values for ITEMVALUE are specifically for use with IPC:

Item	# Type	Description
34	Integer	The current number of writers.
35	Integer	The current number of readers.
49	Logical	The *plabel* of the user's soft interrupt procedure. A value of zero implies that soft interrupts are not being used.

The following intrinsics are not allowed for message files:

FPOINT

FREADSEEK

FUPDATE

FDELETE

FREADDIR

FSPACE

FWRITEDIR

Note

The FSETMODE intrinsic is permitted, but ignored.

EXAMPLES USING MESSAGE FILES

The following programs illustrate the use of IPC via message files. Intrinsics called within the programs manipulate the message files to produce a unidirectional flow of information.

In these two programs, the first is sending information to the second through a message file. The first program, PROC1, reads data from a data file and writes it to MSGFILE2. The second program, PROC2, can then read this data from MSGFILE2 and print it.

When PROC2 finishes reading and printing the data, it writes a message to MSGFILE1 indicating this and terminates. PROC1 reads this message from MSGFILE1 and also terminates. The messages travel among processes and message files as illustrated in Figure 8-1.

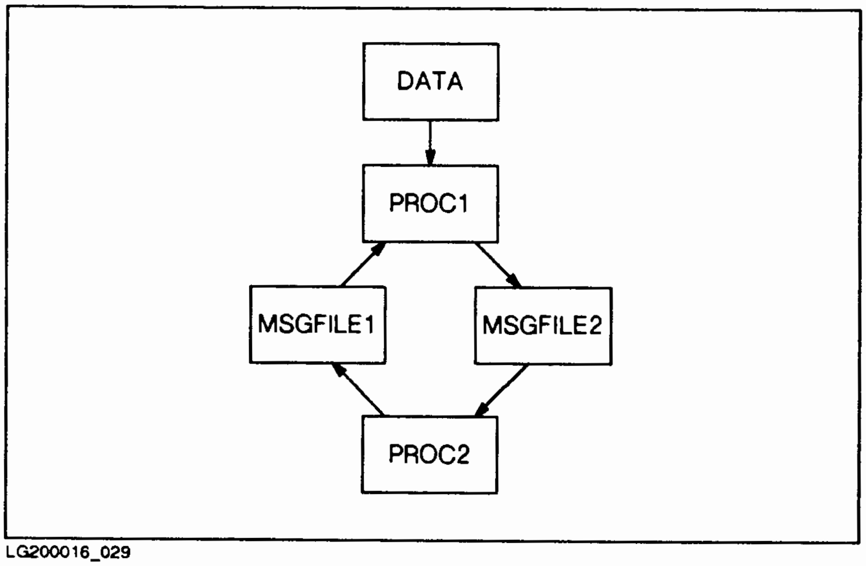
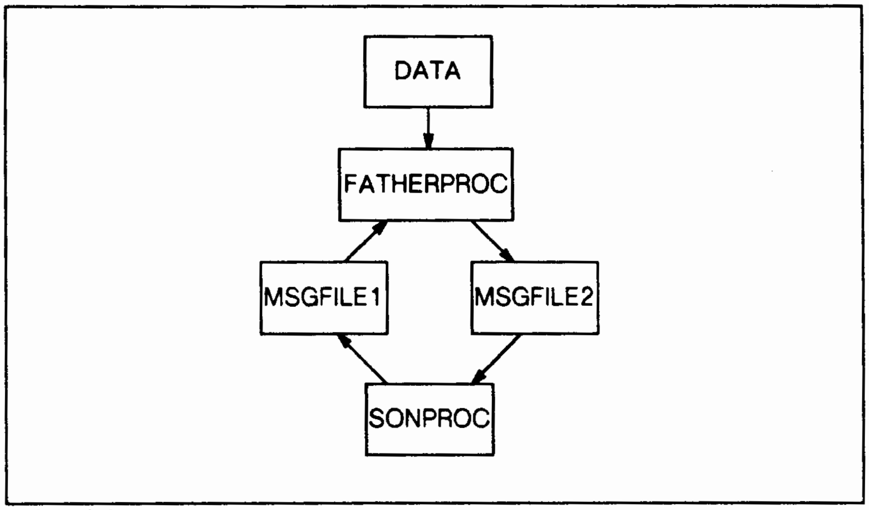


Figure 8-1. Data Paths among Processes and Message Files

```
$CONTROL USLINIT

<<Purpose:>>
<<Read data from a data file and send to another process.>>

BEGIN
  LOGICAL EOF := FALSE;
  INTEGER DATA'FILE, LEN, PIN, IN'FILE, OUT'FILE;

  BYTE ARRAY IN'FILE'NAME (0:8) := "MSGFILE1 ";
  BYTE ARRAY OUT'FILE'NAME (0:8) := "MSGFILE2 ";
  BYTE ARRAY DATA'FILE'NAME (0:8) := "DATA ";
  BYTE ARRAY PRINTPROC (0:8) := "PRNTPROC ";
  ARRAY MESSAGE (0:39);

  INTRINSIC CREATEPROCESS, FCLOSE, FOPEN, FREAD, FWRITE,
       QUITPROG, PRINT, READ;

  <<Create entries for the message files in the directory:>>

  <<Note that IN'FILE'NAME ("MSGFILE1") is opened with FOPTIONs>>
  <<%30004: this indicates a new ASCII message file.>>

  IN'FILE := FOPEN (IN'FILE'NAME, %30004);
  IF < THEN QUITPROG (1);
FCLOSE (IN'FILE, 2, 0); << Save file as session temporary.>>
    IF < THEN QUITPROG (2);

<<Note that OUT'FILE'NAME ("MSGFILE2") is opened with FOPTIONs>>
<<%30004: this indicates a new ASCII message file.>>

OUT'FILE := FOPEN (OUT'FILE'NAME, %30004);
IF < THEN QUITPROG (3);
FCLOSE (OUT'FILE, 2, 0); <<Save file as session temporary>>
  IF < THEN QUITPROG (4);

<<Create and activate the print process:>>

CREATEPROCESS (, PIN, PRINT'PROC)
IF < THEN QUITPROG (5);
```

```
<<Open message file for traffic from print process:>>

<<Note that IN'FILE'NAME ("MSGFILE1") is opened with FOPTIONs>>
<<%106 and AOPTIONs %1100: %106 indicates an old temporary>>
<<ASCII file and %1100 indicates a reader process with>>
<<exclusive access and multi-access capability. MSGFILE1>>
<<has already been designated as a message file. Since>>
<<only one reader and one writer process will be accessing>>
<<the message file, exclusive access mode is specified.>>

IN'FILE := FOPEN (IN'FILE'NAME, %106, %1100);
IF < THEN QUITPROG (7);

<<Open message file for traffic to print process:>>

<<Note that OUT'FILE'NAME ("MSGFILE2") is opened with FOPTIONs>>
<<%106 and AOPTIONs %1101: %106 indicates an old temporary>>
<<ASCII file and %1101 indicates a writer process with>>
<<exclusive access and multi-access capability. MSGFILE2 has>>
<<already been designated as a message file. Since only>>
<<one reader and one writer process will be accessing the>>
<<message file, exclusive access mode is specified.>>

OUT'FILE := FOPEN (OUT'FILE'NAME, %106, %1101);
IF < THEN QUITPROG (8);

<<Open data input file:>>

<<Note that DATA'FILE'NAME ("DATA") is opened with FOPTIONs %3>>
<<and AOPTIONs 0: %3 indicates an old permanent or temporary>>
<<file and 0 indicates read only access. The file system>>
<<will change the FOPTIONs to specify an ASCII file.>>

DATA'FILE := FOPEN (DATA'FILE'NAME, %3, 0);
IF <> THEN QUITPROG (9);
  WHILE NOT EOF DO BEGIN
  LEN ‘s:= FREAD (DATA'FILE, MESSAGE, -80);
  IF < THEN QUITPROG (10);
  IF > THEN EOF := TRUE
  ELSE BEGIN
    FWRITE (OUT'FILE, MESSAGE, -LEN, 0);
    IF <> THEN QUITPROG (11);
    END;
  END << WHILE >>;
  FCLOSE (OUT'FILE, 4, 0); <<No more data to send: EOF>>
     IF < THEN QUITPROG (12);


  FREAD (IN'FILE, MESSAGE, 1); <<Wait for printing process>>
    IF <> THEN QUITPROG (1 <<to finish.>>


  FCLOSE (IN'FILE, 4, 0);
  IF < THEN QUITPROG (14);
  END.

$CONTROL USLINIT

  <<Purpose:>>
  <<Receive data from other process and print it.>>

BEGIN
  LOGICAL EOF := FALSE;
  INTEGER LEN, IN'FILE, OUT'FILE;

  BYTE ARRAY IN'FILE'NAME (0:8) := "MSGFILE2 ";
  BYTE ARRAY OUT'FILE'NAME (0:8) := "MSGFILE1 ";
  ARRAY MESSAGE (0:39);

  INTRINSIC FCLOSE, FOPEN, FREAD, FWRITE, QUITPROG, PRINT;

  <<Open message file for traffic from other process:>>

  <<Note that IN'FILE'NAME ("MSGFILE2") is opened with FOPTIONs>>
  <<%106 and AOPTIONs %1100: %106 indicates an old temporary>>
  <<ASCII file and %1100 indicates a reader process with>>
  <<exclusive access and multi-access capability. MSGFILE2>>
  <<has already been designated as a message file. Since>>
  <<only one reader and one writer process will be accessing>>
  <<the message file, exclusive access mode is specified.>>

  IN'FILE := FOPEN (IN'FILE'NAME, %106, %1100);
  IF < THEN QUITPROG (13);
  <<Open message file for traffic to other process:>>
  <<Note that OUT'FILE'NAME ("MSGFILE1") is opened with FOPTIONs>>
  <<%106 and AOPTIONs %1101: %106 indicates an old temporary>>
  <<ASCII file and %1101 indicates a writer process with>>
  <<exclusive access and multi-access capability. MSGFILE1>>
  <<has already been designated as a message file. Since only>>
  <<one reader and one writer process will be accessing the>>
  <<message file, exclusive access mode is specified.>>

  OUT'FILE := FOPEN (OUT'FILE'NAME, %106, %1101);
  IF < THEN QUITPROG (14);

  WHILE NOT EOF DO BEGIN

   LEN := FREAD (IN'FILE, MESSAGE, &--;80);
    IF < THEN QUITPROG (15);
    IF > THEN EOF := TRUE
    ELSE PRINT (MESSAGE, &--;LEN, 0);
 END << WHILE >>;

<<Now signal other process; we are done.>>

  FCLOSE (OUT'FILE, 4, 0);
  IF < THEN QUITPROG (16);


  FCLOSE (IN'FILE, 4, 0);
  IF < THEN QUITPROG (17);

END .
```

The following two COBOL programs perform the same tasks as the preceding SPL programs. The first program, FATHERPROC, reads data from a data file and writes it to MSGFILE2. The second program, SONPROC, can then read this data from MSGFILE2 and print it. When SONPROC finishes reading and printing the data, it writes a message to MSGFILE1 indicating this and terminates. FATHERPROC" reads this message from MSGFILE1 and also terminates. The messages travel among processes and message files as illustrated in Figure 8-2.



LG200016_030

Figure 8-2. Data Paths among Processes and Message Files

```
$CONTROL USLINIT
IDENTIFICATION DIVISION.
PROGRAM-ID. FATHERPROC.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP3000.
OBJECT-COMPUTER. HP3000.
SPECIAL-NAMES.
CONDITION-CODE IS CC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATA-FILE PIC S9(4) COMP.
01 LEN PIC S9(4) COMP.
01 PIN PIC S9(4) COMP.
01 IN-FILE PIC S9(4) COMP.
01 OUT-FILE PIC S9(4) COMP.
01 IN-FILE-NAME PIC X(9) VALUE "MSGFILE1 ".
01 OUT-FILE-NAME PIC X(9) VALUE "MSGFILE2 ".
01 DATA-FILE-NAME PIC X(5) VALUE "DATA ".
01 PRINTPROC PIC X(9) VALUE "PRNTPROC ".
01 MESSAGE-BUF PIC X(80).
01 EOF-VAR PIC X. 88 EOF VALUE "E".
* ERROR VARIABLES
01 ERROR-BUFFER.
05 FILLER PIC X OCCURS 1 TO 80 TIMES
DEPENDING ON LEN.
01 ERR-NUM PIC S9(4) COMP.
01 FILE-NUM PIC S9(4) COMP.
OI QUIT-PARM PIC S9(4) COMP.
PROCEDURE DIVISION.
MAIN PROCESSING SECTION.
$DEFINE %QUITPROG= QUITPROG
MOVE !1 TO QUIT-PARM QUITPROG
MOVE !2 TO FILE-NUM QUITPROG
PERFORM PRINT-ERROR# QUITPROG
DRIVER-PARA.
PERFORM INIT-PARA.
MOVE "F" TO EOF-VAR.
PERFORM LOAD-PARA UNTIL EOF.
PERFORM CLOSE-PARA.
STOP RUN.
```

```
*
* Create entries for the message files in the directory.
*
* Note that IN-FILE-NAME "MSGFILE1") is opened with FOPTIONs
* %30004: this indicates a new ASCII message file.
*
INIT-PARA.
      CALL INTRINSIC "FOPEN"
    USING IN-FILE-NAME %30004
    GIVING IN-FILE,
      IF CC NOT = 0
  %QUITPROG(1#,IN-FILE#).
 CALL INTRINSIC "FCLOSE" USING IN-FILE %2 %0.
 IF CC NOT = 0
  %QUITPROG(2#,IN-FILE#).
*
* Note that OUT-FILE-NAME ("MSGFILE2") is opened with FOPTIONs
* %30004: this indicates a new ASCII message file.
*
      CALL INTRINSIC "FOPEN"
      USING OUT-FILE-NAME %30004
      GIVING OUT-FILE,
       IF CC NOT = 0
  %QUITPROG(3#,OUT-FILE#).
       CALL INTRINSIC "FCLOSE" USING OUT-FILE %2 %0.
       IF CC NOT = 0
  %QUITPROG(4#,OUT-FILE#).
*
* Create and activate the print process.
*
       CALL INTRINSIC "CREATEPROCESS" USING PIN PRINTPROC.
       IF CC NOT = 0
   %QUITPROG(5#,-1#).
*
* Open message file for traffic from print process,
*
* Note that IN-FILE-NAME ("MSGFILE1") is opened with FOPTIONs
* %106 and AOPTIONs %1100: %106 indicates an old temporary
* ASCII file and %1100 indicates a reader process with
* exclusive access and multi-access capability. MSGFILE1 has
* already been designated as a message file.
* Since only one reader and one writer process will
* be accessing the message file, exclusive access
* mode is specified.
     CALL INTRINSIC "FOPEN"
     USING IN FILE-NAME %106 %1100
     GIVING IN-FILE.
      IF CC NOT = 0
 %QUITPROG (7#,IN-FILE#).
```

```
*
* Open message file for traffic to print process.
*
* Note that OUT-FILE-NAME ("MSGFILE2") is opened with FOPTIONs
* %106 and AOPTIONs %1101: %106 indicates an old temporary
* ASCII file and %1101 indicates a writer process with
* exclusive access and multi-access capability. MSGFILE2 has already
* been designated as a message file. Since only one reader and
* one writer process will be accessing the message file,
* exclusive access mode is specified.
*
  CALL INTRINSIC "FOPEN"
    USING OUT-FILE-NAME %106 %1101
    GIVING OUT-FILE.
   IF CC NOT = 0
 %QUITPROG(8#,OUT-FILE#).
*
* Open data input file.
*
*Note that DATA-FILE-NAME ("DATA") is opened with FOPTIONs %3
*and AOPTIONs 0: %3 indicates an old permanent or temporary
*file and 0 indicates read only access. The file system will
*change the FOPTIONs to specify an ASCII file.
*
  CALL INTRINSIC "FOPEN"
    USING DATA-FILE-NAME %3 %0
    GIVING DATA-FILE.
    IF CC NOT = 0
 %QUITPROG(9 # , DATA-FILE # ) .
*
* Load input to message file.
*
LOAD-PARA.
 CALL INTRINSIC "FREAD"
       USING DATA-FILE MESSAGE-BUF &--;80
       GIVING LEN.
 IF CC NOT = 0
    IF CC LESS THAN 0 THEN
     %QUITPROG(10#,DATA-FILE#)
    ELSE
      MOVE "E" TO EOF-VAR
 ELSE
    COMPUTE LEN = &--; LEN
    CALL INTRINSIC "FWRITE"
    USING OUT-FILE MESSAGE-BUF LEN %0
    IF CC NOT = 0
     %QUITPROG(11#,OUT-FILE#).
CLOSE-PARA.
 CALL INTRINSIC "FCLOSE" USING OUT-FILE %4 %0.
 IF CC NOT =00
    %QUITPROG(12#,OUT-FILE#),

*
* Wait for print to finish.
*
 CALL INTRINSIC "FREAD" USING IN-FILE MESSAGE-BUF %1.
 IF CC < 0
 %QUITPROG(13#,IN-FILE#).
 CALL INTRINSIC "FCLOSE" USING IN-FILE %4 %0.
 IF CC NOT = 0
 %QUITPROG(14#,IN-FILE#).
*
* General error routine.
*
PRINT-ERROR SECTION.
WHAT-TYPE.

       IF FILE-NUM IS NOT NEGATIVE THEN
  CALL INTRINSIC "FCHECK" USING FILE-NUM ERR-NUM
     MOVE 80 TO LEN
    CALL INTRINSIC "FERRMSG" USING ERR-NUM ERROR-BUFFER LEN
     DISPLAY ERROR-BUFFER.
  IF QUIT-PARM IS NOT NEGATIVE THEN
     CALL INTRINSIC "QUITPROG" USING QUIT-PARM.
$CONTROL USLINIT
IDENTIFICATION DIVISION.

PROGRAM-ID.
SONPROC.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP3000.
OBJECT-COMPUTER. HP3000.
SPECIAL-NAMES.
CONDITION-CODE IS CC.
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 LEN PIC S9(4) COMP.
01 IN-FILE PIC S9(4) COMP.
01 OUT-FILE PIC S9(4) COMP.
01 IN-FILE-NAME PIC X(9) VALUE "MSGFILE2 ".
01 OUT-FILE-NAME PIC X(9) VALUE "MSGFILE1 ".
01 MESSAGE-BUF PIC X(80).
01 EOF-VAR PIC X.
 88 EOF VALUE "E".
* Error variables.
01 ERROR-BUFFER.
    05 FILLER PIC X OCCURS 1 TO 80 TIMES
        DEPENDING ON LEN.
01 ERR-NUM PIC S9(4) COMP.
01 FILE-NUM PIC S9(4) COMP.
01 QUIT-PARM PIC S9(4) COMP.
PROCEDURE DIVISION.
MAIN-PROCESSING SECTION.
$DEFINE %QUITPROG= QUITPROG
      MOVE !1 TO QUIT-PARM QUITPROG
MOVE !2 TO FILE-NUM QUITPROG
      PERFORM PRINT-ERROR# QUITPROG

DRIVER-PARA.

      PERFORM OPEN-PARA.
      MOVE "F" TO EOF-VAR.
      PERFORM READ-PARA UNTIL EOF.
      PERFORM CLOSE-PARA.
      STOP RUN.
*
* Open message file for traffic from other process,
*
* Note that IN-FILE-NAME ("MSGFILE2") is opened with FOPTIONs
* %106 and AOPTIONs %1100: %106 indicates an old temporary
* ASCII file and %1100 indicates a reader process with
* exclusive access and multi-access capability. MSGFILE2 has
* already been designated as a message file. Since only one
* reader and one writer process will be accessing the message
* file, exclusive access mode is specified.
*
OPEN-PARA.
      CALL INTRINSIC "FOPEN"
      USING IN-FILE-NAME %106 %1100
      GIVING IN-FILE.
       IF CC NOT = 0
   %QUITPROG(15#,IN-FILE#) .
```

```
*
* Open message file for traffic to other process.
*
* Note that OUT-FILE-NAME ("MSGFILE1") is opened with FOPTIONs
* %106 and AOPTIONs %1101: %106 indicates an old temporary
* ASCII file and %1101 indicates a writer process with
* exclusive access and multi-access capability. MSGFILE1 has already
* been designated as a message file. Since only one reader and
* one writer process will be accessing the message file,
* exclusive access mode is specified,
*
     CALL INTRINSIC "FOPEN"
     USING OUT-FILE-NAME %106 %1101
     GIVING OUT-FILE.
      IF CC NOT = 0
  %QUITPROG (16#,OUT-FILE#) .
*
* Read messages from message file.
*
READ-PARA.
       CALL INTRINSIC "FREAD"
        USING IN-FILE MESSAGE-BUF -80
        GIVING LEN.
       IF CC NOT = 0
   IF CC LESS THAN 0 THEN
     %QUITPROG (17# , IN-FILE#)
    ELSE
      MOVE "E" TO EOF-VAR
*
* Print message out.
*
    ELSE
       COMPUTE LEN = -- LEN
       CALL INTRINSIC "PRINT"
        USING MESSAGE-BUF LEN %0
    IF CC NOT = 0
       %QUITPROG(18#,2#).
*
* Now signal the other process; we are done.
*
CLOSE-PARA.
 CALL INTRINSIC "FCLOSE" USING OUT-FILE %4 %0.
 IF CC NOT = 0
   %QUITPROG(19#,OUT-FILE#).
 CALL INTRINSIC "FCLOSE" USING IN-FILE %4 %0.
 IF CC NOT = 0
   %QUITPROG(20#,IN-FILE#).
```

```
*
* General error routine.
*

PRINT-ERROR SECTION.
WHAT-TYPE.

 IF FILE-NUM IS NOT NEGATIVE THEN
   CALL INTRINSIC "FCHECK" USING FILE-NUM ERR-NUM
     MOVE 80 TO LEN
    CALL INTRINSIC "FERRMSG" USING ERR-NUM ERROR-BUFFER LEN
     DISPLAY ERROR-BUFFER.
  IF QUIT-PARM IS NOT NEGATIVE THEN
     CALL INTRINSIC "QUITPROG" USING QUIT-PARM.
```

Circular Files

Circular files are wrap-around structures which behave as standard sequential files until they are full. As records are written to a circular file, they are appended to the end of the file. When the file is filled, the next record added causes the block at the front of the file to be deleted and all other blocks to be logically shifted toward the front 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 front (oldest record remaining) 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 may have 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 in the *AOPTIONs* that it will be a circular file, the 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 LISTF,2 command, circular files will be 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        128W   FBO    0    1023   1        12   1  8
```

Features of Intrinsics for Circular Files

Most intrinsics treat circular files the same way they treat regular disk files, but some have special features which apply specifically to circular files. Most of these features are found in FOPEN, but a few other intrinsics are also affected.

Parameters not mentioned in the following descriptions retain their normal range of values and their normal default values.

FOPEN

FOPTIONs: (2:3) - File type. Determines the type of file to create. If the file is old, this field is ignored.

000 -	Ordinary file
001 -	KSAM file
010 -	Relative I/O file
100 -	Circular file
110 -	Message file

AOPTIONs: (5:2) - Multi-access mode. This feature permits processes located in different jobs or sessions to open the same file.

00 - No multi-access. For a writer, the file system changes this value to a 2 for global multi-access.

01 - Only intra-job multi-access allowed. This is the same as specifying the MULTI option in a FILE command.

10 - Interjob multi-access allowed. This is the same as specifying the GMULTI option in a FILE command.

11 - Undefined. If this is specified, the FOPEN will be rejected with an error code of 40: ACCESS VIOLATION.

(7:1) - Inhibit buffering. Reader processes may open circular files with either the `BUF` or `NOBUF` Option. For write access to circular files, the file system sets this bit off.

Note: Readers may open a circular file with `NOBUF` if they are in copy mode. The `NOBUF` bit determines whether the file will be read record by record or block by block:

0 - Read by logical record

1 - Read by physical block

(8:2) - Exclusive. The values for this field are the same as for any standard disk file, but they have different meanings for the readers and writers of a circular file:

	Changed To:	
User Value	**READER**	**WRITER**
EXCLUSIVE	EXCLUSIVE	EXCLUSIVE
SEMI	SHARE	EXCLUSIVE
SHARE	SHARE	SHARE
Default	SHARE	EXCLUSIVE

For readers, SHARE means “allow other readers”. For writers, SHARE means “allow other writers”.

(11:1) - Multi-record. When a reader specifies this option, the file will be accessed`NOBUF`. For writers, this bit is set to zero.

(12:4) - Access type. These bits specify whether the user will be a reader or a writer process.

0000- READ access only.

0001- WRITE access only. If this is the first accessor to the file, then the file's contents are purged. If this is not the first accessor to the file, the access type is set to APPEND.

0010- WRITE SAVE access. Set to APPEND access.

0011- APPEND access only.

Note: Circular files allow variable-length records with append access.

Any other access types are invalid.

`FILESIZE` The number of records is rounded up to completely fill the last block.

FWRITE

This intrinsic logically appends the user's record to the end of the file. If the file is full, the first block is deleted, the remaining blocks are logically shifted to the file's head, and the new record is appended to the end of the file.

FCLOSE

For circular files, deletion of disk space beyond the end-of-file is not allowed.

Certain intrinsics are not allowed when circular files are used. These intrinsics are listed in Table 8-2.

Table 8-2. Intrinsics not Permitted with Circular Files

Not permitted for READ access	Not permitted for WRITE access
FUPDATE	FUPDATE
FDELETE	FDELETE
FWRITEDIR	FWRITEDIR
FWRITE	FREAD
	FREADDIR
	FREADSEEK
	FPOINT
	FSPACE

SOFTWARE INTERRUPTS

The software interrupt facility enables you to perform `FREAD` and `WRITE` completion processing with your own interrupt procedures.

A call to `FREAD` or `FWRITE` is necessary to initiate the I/O request. Both of these intrinsics will return control to your program as soon as the request has begun. When the operation completes, your program is trapped (or interrupted) to a procedure of your choice. This procedure performs whatever processing is necessary and then exits back to your mainline program.

Initially, software interrupts are disabled for your programs. To enable soft interrupts, use the `FINTSTATE` intrinsic with a value of `TRUE`, as follows:

```
VALUE:=FINSTATE(TRUE);
```

The `FINTSTAT` intrinsic called with a value of `FALSE` will inhibit soft interrupts. MPE will inhibit soft interrupts just before entering an interrupt procedure. This is done to prevent unwanted nesting of the interrupt procedures. Use the FINTEXIT intrinsic to return from an interrupt procedure. It will re-enable soft interrupts just before it exits.

```
PROCEDURE INTERRUPTPROC(FILENUM);
  VALUE FILENUM;
  INTEGER FILENUM;
    BEGIN
     :
   FINTEXIT;
    END;
```

Software interrupts are automatically inhibited just before a CONTROL-Y trap procedure. The trap procedure may elect to allow soft interrupts by calling the FINTSTATE intrinsic. If it does not call FINTSTATE, the RESETCONTROL intrinsic will restore the process' interrupt state to its pre-CONTROL-Y value.

When you have enabled software interrupts for your program, you *arm* them for a particular file by specifying the interrupt procedure's *plabel* in an FCONTROL 48. Calling FCONTROL 48 with a parameter of zero will disarm the software interrupt mechanism so the file can be accessed in the normal manner.

Note

The FFILEINFO intrinsic may be used to return the *plabel* of the interrupt handler. FFILEINFO 49 will return the *plabel* as an integer value: if it returns a value of zero, no interrupt handler has been armed.

After an interrupt has been received, an IODONTWAIT must be issued against the file to complete the request. Your interrupt handling procedure will usually issue the IODONTWAIT before it handles the interrupt completion processing.

Note

Only message files allow soft interrupts.

No more than one uncompleted FREAD or FWRITE may be outstanding for a particular file. Any additional FREADS or FWRITES will be rejected.

The interrupt will not occur while you are executing within MPE. That is, while you are processing an MPE intrinsic or procedure. Exceptions are the PAUSE, PAUSEX, and some IOWAIT intrinsics will allow the interrupt. When the interrupt procedure exits, interrupts are reenabled.

The timer in PAUSE will be restarted from the beginning. The timer in PAUSEX Will be restarted where it left off. Only IOWAITS against a message file, or a general IOWAIT (IOWAIT(0);) can be interrupted. An IOWAIT against a specific non-message file cannot be interrupted.

Software interrupts may not be used with remote files.

An uncompleted FREAD or FWRITE request may be aborted by issuing an FCONTROL 43 (abort NOWAIT I/O).

Example Use of Software Interrupts

The two primary advantages of software interrupts are that they are handled transparently to the process' mainline code and that they are given real time response by the target process. This example uses both advantages in the control of a multiprocess transaction processing system.

The three types of processes in the system are terminal processes, function processes, and supervisor processes.

Terminal Processes.

Each terminal has its own private terminal process. These processes perform some pre-editing of each transaction and then send it to the proper function process.

Function Processes

These are expert in some particular aspect of the system. For example, one for payroll, one for accounts receivable and so on. They accept input from any of the terminal processes, using message files.

Supervisor Process

There is only one supervisor process. It accepts commands from its terminal and then *forces* the appropriate terminal/function process to execute the command. Examples of the commands would be:

- Report process status and/or run-time statistics.
- Set checkpoints, change files, etc.
- Enter DEBUG.
- Terminate gracefully.

To get the attention of the target process, the supervisor process need only send information to the target process' "control" message file. The target process has already enabled soft interrupts on the file, so the supervisor process' FWRITE will soft interrupt it.

This is a function/terminal process code fragment that enables soft interrupts:

```
          CONTROLFILE:=FOPEN( ... );
          INTADDRESS:=@INTHANDLER;
          FCONTROL(CONTROLFILE,48,INTADDRESS);
          IF < > THEN ERROR(CONTROLFILE);

          FREAD(CONTROLFILE,DUMMY,CMDLEN);
          IF < > THEN ERROR(CONTROLFILE);
          FINTSTATE(TRUE);
```

This is a function/terminal process interrupt handler:

```
          PROCEDURE INTHANDLER(FILENUM);
          VALUE FILENUM;
          INTEGER FILENUM;
     BEGIN
     ARRAY CMD (0 : CMDLEN) , REPLY (0 : REPLYLEN) ;
     INTEGER REPLYSIZE;

          IODONTWAIT(FILENUM,CMD);
          IF < > THEN ERROR(FILENUM);
          CASE CMD OF
     BEGIN << PERFORM COMMAND, FORM REPLY >>
       :
         END ;
        FWRITE(REPLYFILE, REPLY,REPLYSIZE,0);
        IF < > THEN ERROR(REPLYFILE);
        FINTEXIT;
        END; << INTHANDLER >>
```

The validity of an interrupt procedure depends on the code domain of your code executing mode (privileged or non-privileged) and on the code domain of the *plabel* and the mode (privileged or non-privileged). (See Table 8-3.) The code domains are:

PROG	(User Program)
GSL	(Group SL)
PSL	(Public SL)
SSL	(System SL,non-MPE segments)
MPESSL	(System SL, MPE segments)

Table 8-3. Interrupt Procedure Code Domain Requirements

WHEN THE CODE OF THE CALLER IS:	THE PLABEL:
Non-privileged in PROG, GSL, or PSL	Must be non-privileged in PROG, GSL, or PSL.
Privileged in PROG, GSL, or PSL.	May be privileged or non-privileged in PROG, GSL, or PSL.
Privileged or non-privileged in SSL.	May be in any non-MPESSL segment.

Section Divider

9. Magnetic Tape Considerations

9

Magnetic Tape Considerations

Magnetic Tape Considerations

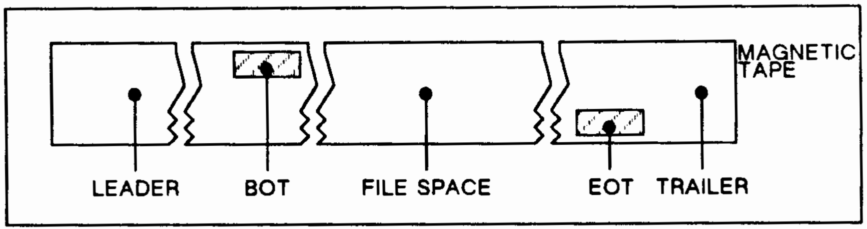
This chapter describes the matters you should keep in mind when you work with your magnetic tape files.

Note

Serial disk files are very similar to magnetic tape files. Unless otherwise noted, information in this chapter applies to serial disks as well as to magnetic tape.

Beginning of Tape (BOT) and End of Tape (EOT) Markers

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). 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 start or end of tape positions:



LG200016_031

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 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 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).

If the magnetic tape is labeled (as specified in the 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 volume (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.

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.

FREADBACKWARD

If the BOT marker is encountered, a CCG condition code is returned. However, when reading a labeled magnetic tape file that spans more than one volume, CCG is not returned when the BOT marker is encountered; instead, CCG is returned at the actual beginning of the file, with a transmission log of 0 if an attempt is made to read past the beginning of the file.

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 will be 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 which 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 which 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 2 (%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 which originally created (wrote) the tape.

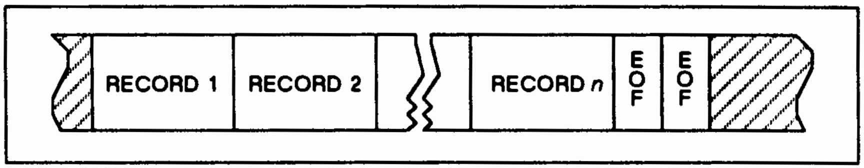
A program which needed to create a multi-volume (multiple reel) tape file would normally write tape records until the status returned from FWRITE indicated an EOT condition. Writing could be continued in a limited manner to reach a logical point to break the file. Then several file marks and a trailing tape label would typically be added, the tape rewound, another reel mounted, and the data transfer continued. The program designed to read such a multi-volume file must expect to find and check for the EOF and the 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.

End-of-File Marks on Magnetic Tape

An FWRITE to magnetic tape, followed by any intrinsic call which reverses tape motion (for example, backspace a record, backspace a file, or rewind), causes the file system to write an EOF mark before initiating the reverse motion.

For example, if a user program has just written several data records to magnetic tape and writes a file mark, rewinds the tape, and closes the file, the tape file will be terminated by two file marks (EOF).

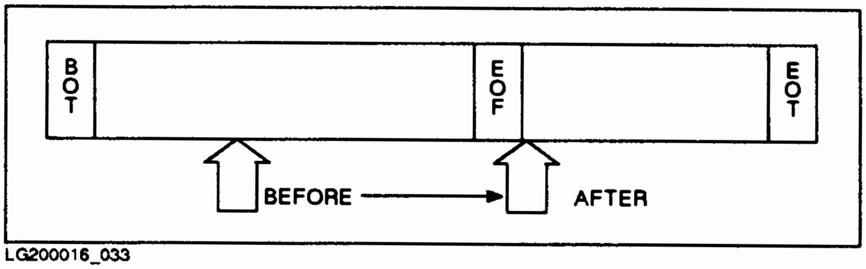
The first of these was requested by the user by calling FCONTROL to write an EOF, and the second was provided by the system because the direction of tape motion had been reversed after a write (rewind):



LG200016_032

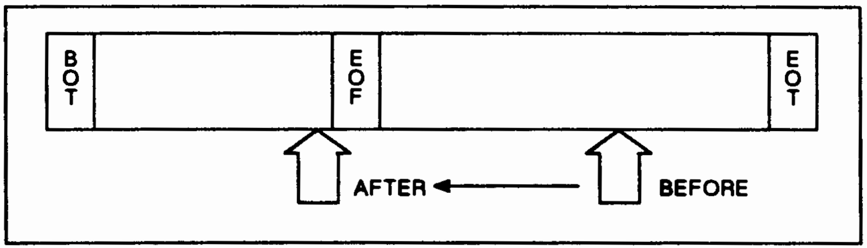
Spacing File Marks

When you space forward to a tape mark (EOF), the tape recording heads have just read the EOF and are positioned beyond it:



LG200016_033

When you space backward to a tape mark (EOF), the mark is recognized as the tape travels in the reverse direction. The tape heads then are left positioned just in front of the EOF that was read:



LG200016_034

When FREAD has found a logical file mark and returned a condition code of CCG, the EOF mark has been read and the tape heads are positioned immediately following the mark (similar to space forward to tape mark above).

Using the FCLOSE Intrinsic with Magnetic Tape

The operation of the FCLOSE intrinsic as used with unlabeled magnetic tape is outlined in the flowchart of Figure 9-1.

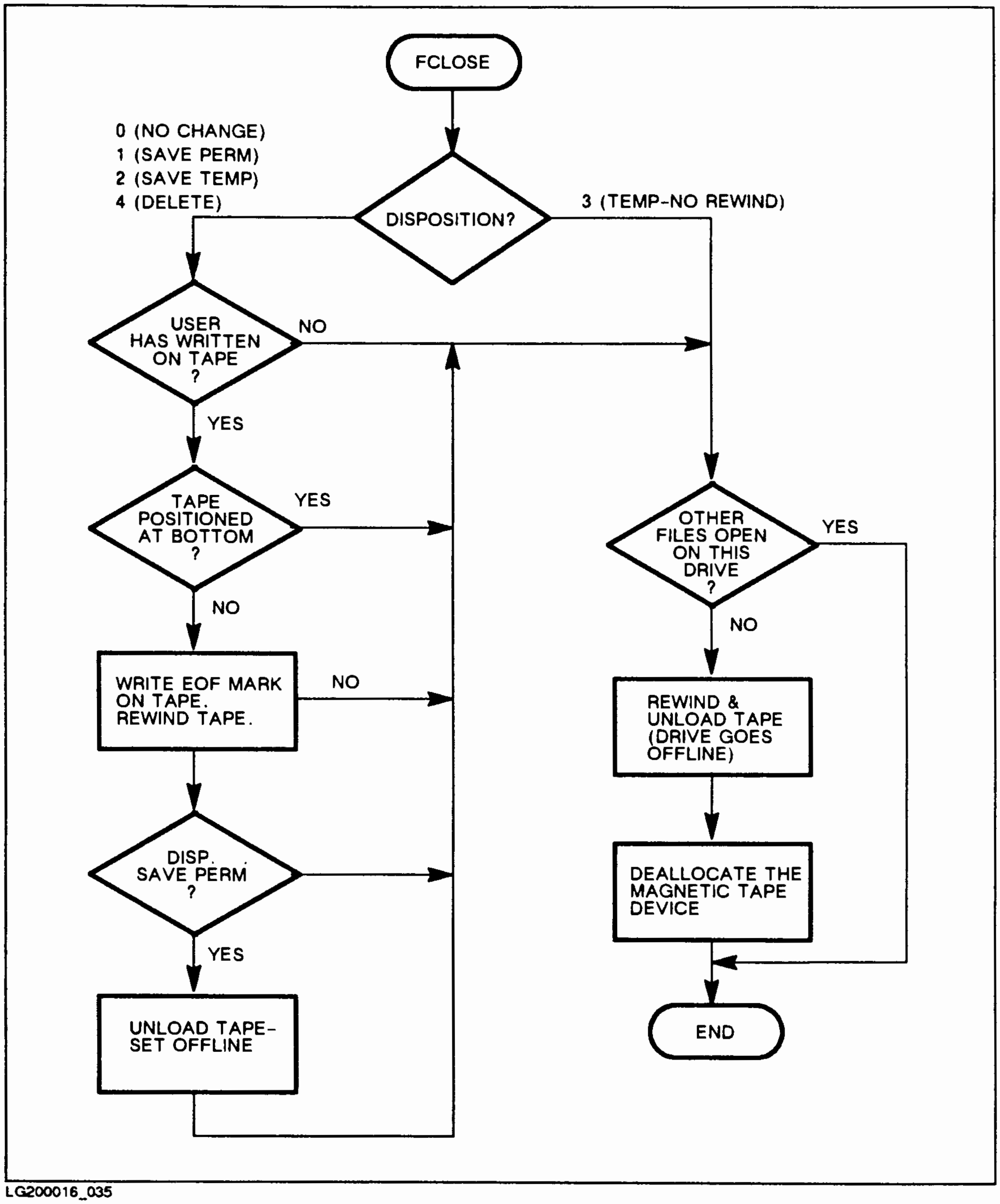

FCLOSE
0 (NO CHANGE)
1 (SAVE PERM)
2 (SAVE TEMP)
4 (DELETE)
DISPOSITION?
3 (TEMP-NO REWIND)
USER HAS WRITTEN ON TAPE ?
NO
YES
TAPE POSITIONED AT BOTTOM ?
YES
NO
WRITE EOF MARK ON TAPE. REWIND TAPE.
NO
DISP. SAVE PERM ?
YES
UNLOAD TAPE-SET OFFLINE
OTHER FILES OPEN ON THIS DRIVE ?
YES
NO
REWIND & UNLOAD TAPE (DRIVE GOES OFFLINE)
DEALLOCATE THE MAGNETIC TAPE DEVICE
END
LG200016_035

Figure 9-1. Using the FCLOSE Intrinsic with Unlabeled Magnetic Tape

Note that a tape closed with the temporary no-rewind disposition will be rewound and unloaded if certain additional conditions are not met. It is possible for a single process to FOPEN a magnetic tape device using a device class and later FOPEN the same device again using its logical device number. This may be done in such a manner that both magnetic tape files are open concurrently. The second FOPEN does not require any operator intervention (for example, for device allocation). When FOPEN/FCLOSE calls are arranged in a nested fashion, tape files may be closed without deallocating the physical device, as follows:

```
[FOPEN         Allocate Tape
[
[
[  [FOPEN
[  [
[  [FCLOSE
[
[              Tape Remains Allocated
[
[  [FOPEN
[  [
[  [FCLOSE
[
[FCLOSE        Deallocated Tape
[
```

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

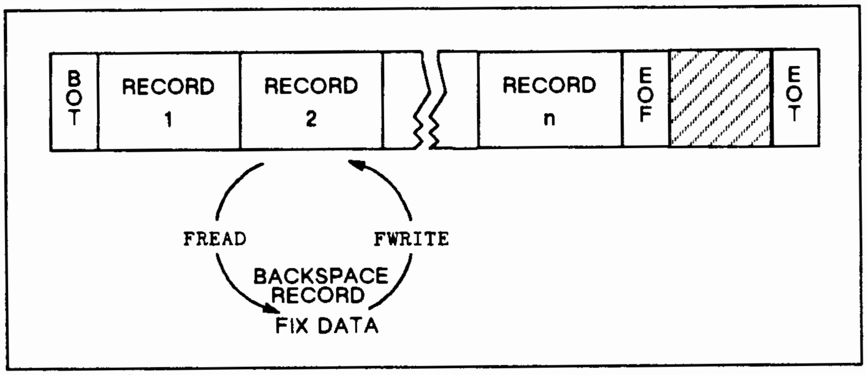
Note that when a temporary no-wind tape is deallocated, the file system has not placed an end-of-file mark at the end of the data file.

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 volume set.

Updating Magnetic Tape Files

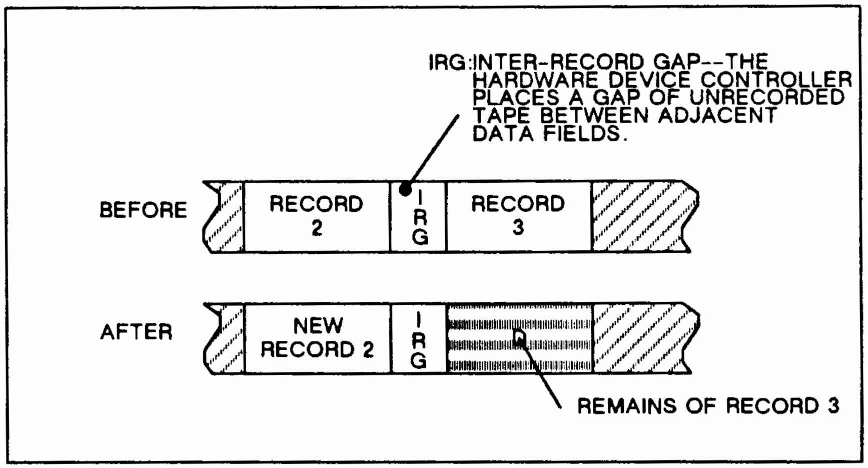
As a physical data storage device, magnetic tape is not designed to enable the replacement of a single record in an existing file. An attempt to perform this type of operation will cause problems in maintaining the integrity of records on the tape. Magnetic tape files, therefore, should not be maintained (updated) on an individual record basis but should be updated during copy operations from one file to another.

As an example of the type of problems that can occur, consider the results of attempting to read a tape record, modify its data, backspace the tape, and overwrite the original record:



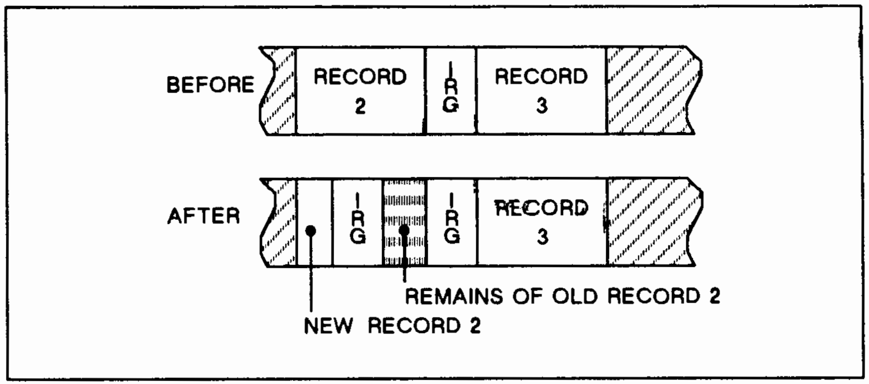
LG200016_036

If the replacement differed at all in size from the original record, the result would not simply be an update of the record. A replacement record of greater length than the original record would overwrite (destroy) a portion of the next record on the tape, as shown below:


IRG:INTER-RECORD GAP--THE HARDWARE DEVICE CONTROLLER PLACES A GAP OF UNRECORDED TAPE BETWEEN ADJACENT DATA FIELDS.
BEFORE
RECORD 2
IRG
RECORD 3
AFTER
NEW RECORD 2
IRG
REMAINS OF RECORD 3

LG200016_037

On the other hand, if the length of the replacement record is less than that of the original record, a portion of the original record will still remain on the tape as shown below:



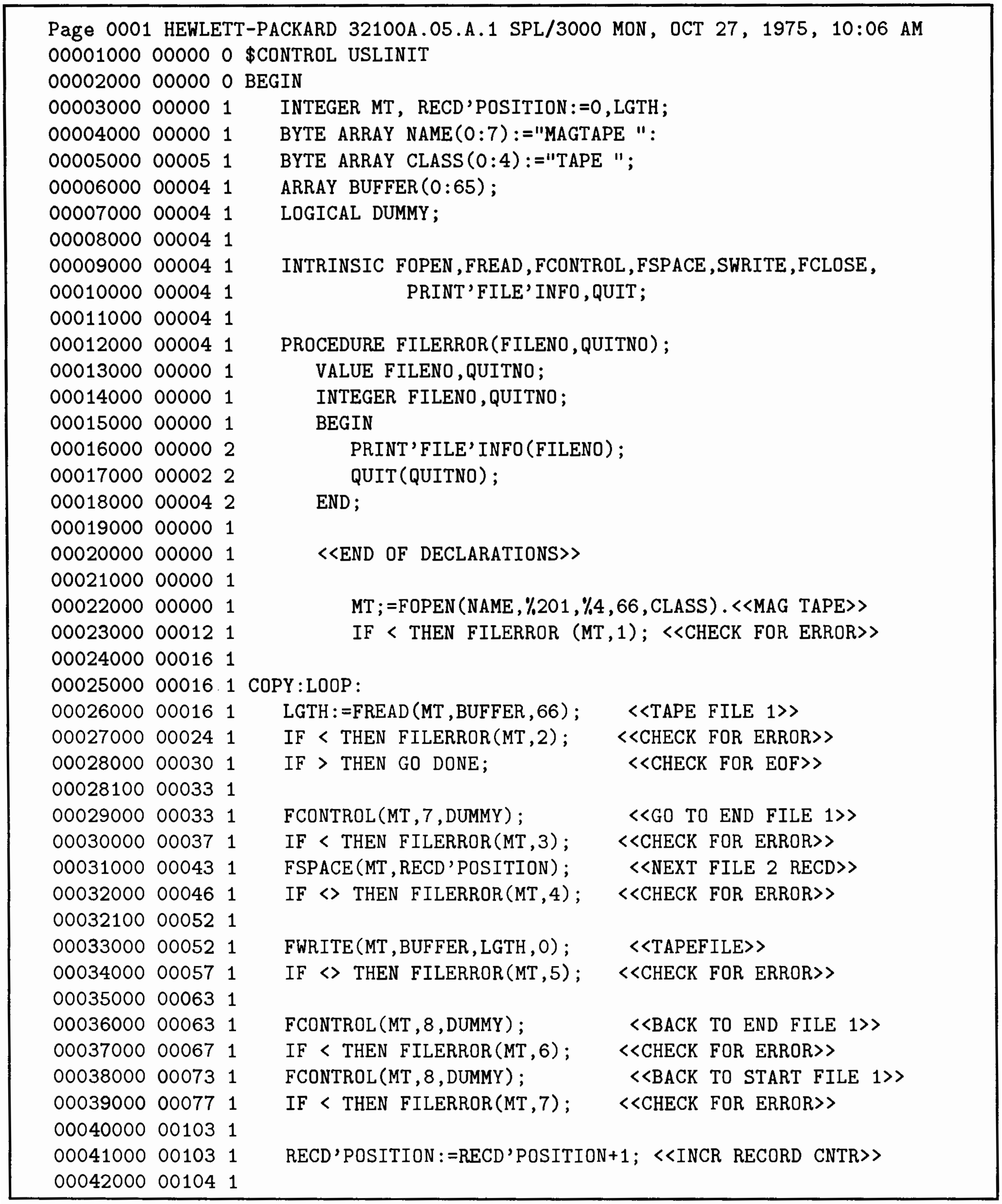
LG200016_038

In either of the two cases shown, the partial records remaining would cause magnetic tape read errors and would create problems in subsequent processing of the tape file.

Even with replacement records of the same size as the original records, errors can result. Mechanical and timing variations from one magnetic tape drive to another can create substantial differences in the actual length of tape records containing the same amount of data. Magnetic tape standards, for example, permit the inter-record gap (IRG) to vary in length from 0.5 to 0.7 inches. Similar variations may occur to a lesser extent in the spacing of the actual data bytes ·recorded. In short, the variation of a number of hardware factors which are beyond the user's control can affect the physical length of the tape records written. For this reason, always update your tape files during copy operations from one tape to another.

Reading and Writing an Unlabeled Magnetic Tape File

Figure 9-2 contains a program that copies an unlabeled magnetic tape file into another file on the same reel of tape.



```
Page 0001 HEWLETT-PACKARD 32100A.05.A.1 SPL/3000 MON, OCT 27, 1975, 10:06 AM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1    INTEGER MT, RECD'POSITION:=0,LGTH;
00004000 00000 1    BYTE ARRAY NAME(0:7):="MAGTAPE ":
00005000 00005 1    BYTE ARRAY CLASS(0:4):="TAPE ";
00006000 00004 1    ARRAY BUFFER(0:65);
00007000 00004 1    LOGICAL DUMMY;
00008000 00004 1
00009000 00004 1    INTRINSIC FOPEN,FREAD,FCONTROL,FSPACE,SWRITE,FCLOSE,
00010000 00004 1               PRINT'FILE'INFO,QUIT;
00011000 00004 1
00012000 00004 1    PROCEDURE FILERROR(FILENO,QUITNO);
00013000 00000 1       VALUE FILENO,QUITNO;
00014000 00000 1       INTEGER FILENO,QUITNO;
00015000 00000 1       BEGIN
00016000 00000 2          PRINT'FILE'INFO(FILENO);
00017000 00002 2          QUIT(QUITNO);
00018000 00004 2       END;
00019000 00000 1
00020000 00000 1       <<END OF DECLARATIONS>>
00021000 00000 1
00022000 00000 1          MT;=FOPEN(NAME,%201,%4,66,CLASS).<<MAG TAPE>>
00023000 00012 1          IF < THEN FILERROR (MT,1); <<CHECK FOR ERROR>>
00024000 00016 1
00025000 00016 1 COPY:LOOP:
00026000 00016 1    LGTH:=FREAD(MT,BUFFER,66);    <<TAPE FILE 1>>
00027000 00024 1    IF < THEN FILERROR(MT,2);    <<CHECK FOR ERROR>>
00028000 00030 1    IF > THEN GO DONE;            <<CHECK FOR EOF>>
00028100 00033 1
00029000 00033 1    FCONTROL(MT,7,DUMMY);         <<GO TO END FILE 1>>
00030000 00037 1    IF < THEN FILERROR(MT,3);    <<CHECK FOR ERROR>>
00031000 00043 1    FSPACE(MT,RECD'POSITION);     <<NEXT FILE 2 RECD>>
00032000 00046 1    IF <> THEN FILERROR(MT,4);   <<CHECK FOR ERROR>>
00032100 00052 1
00033000 00052 1    FWRITE(MT,BUFFER,LGTH,0);     <<TAPEFILE>>
00034000 00057 1    IF <> THEN FILERROR(MT,5);   <<CHECK FOR ERROR>>
00035000 00063 1
00036000 00063 1    FCONTROL(MT,8,DUMMY);         <<BACK TO END FILE 1>>
00037000 00067 1    IF < THEN FILERROR(MT,6);    <<CHECK FOR ERROR>>
00038000 00073 1    FCONTROL(MT,8,DUMMY);         <<BACK TO START FILE 1>>
00039000 00077 1    IF < THEN FILERROR(MT,7);    <<CHECK FOR ERROR>>
00040000 00103 1
00041000 00103 1    RECD'POSITION:=RECD'POSITION+1; <<INCR RECORD CNTR>>
00042000 00104 1
```

Figure 9-2. Unlabeled Magnetic Tape Example

The FOPEN intrinsic call:

```
   MT:=FOPEN(NAME,%201,%4,66,CLASS);
```

opens the magnetic tape file. The parameters specified are:

formaldesignator MAGTAPE, which is contained in the byte array NAME.

foptions %201, for which the bit pattern is as follows:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 Bits

0 0 0 0 0 0 0 0 1 0  0  0  0  0  0  1 Binary

     2     0      1          Octal
```

The above bit pattern specifies the following file options:

Domain: Old permanent file. Bits(14:2) = 01.
ASCII/Binary: Binary. Bit(13:1) = 0.
Default Designator: Same as formal file designator.
Bits(10:3) = 000.
Record Format: Undefined length. Bits(8:2) = 10.

aoptions %4, for which the bit pattern is as follows:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 Bits

0 0 0 0 0 0 0 0 0 0  0  0  0  1  0  0 Binary

   4    Octal
```

The above bit pattern specifies the following access option:

Access Type: Input/output access. Bits (12:4) = 0100.

recsize 'f' words.

device TAPE, contained in the byte array CLASS.

All other parameters are omitted from the FOPEN intrinsic call.

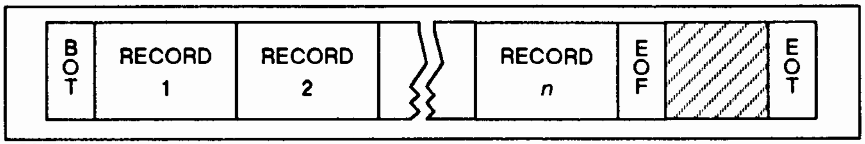
Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable MT.

The statement:

```
   IF<THEN FILERROR(MT,1);
```

checks the condition code and, if it is CCL, calls the error-check procedure FILERROR. The FILERROR procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FOPEN, then aborts the program's process.

The tape format before the copy operation is started is:



LG200016_040

The statement:

```
LGTH:=FREAD(MT,BUFFER,66);
```

reads a record from the file designated by MT and transfers this record to BUFFER. The statement reads up to 66 words from the record, then returns a positive value to LGTH indicating the actual length of the information transferred.

The statement:

```
FCONTROL(MT,7,DUMMY);
```

spaces forward to the EOF tape mark (the end of the file). As you recall from paragraph "Spacing File Marks", the recording head actually is positioned slightly beyond the EOF file mark. Now the statement:

```
FSPACE(MT,RECD'POSITION);
```

spaces the tape to the point where the first record (RECD'POSITION = 0, see statement number 3 in the program) of the second file is to begin. The statement:

```
FWRITE(MT,BUFFER,LGTH,0);
```

writes the record contained in the array BUFFER into this record.

The statement:

```
FCONTROL(MT,8,DUMMY);
```

spaces back to the end of file 1 (the EOF mark) and the statement:

```
FCONTROL(MT,8,DUMMY);
```

then spaces back to the next tape mark (the start of file 1).

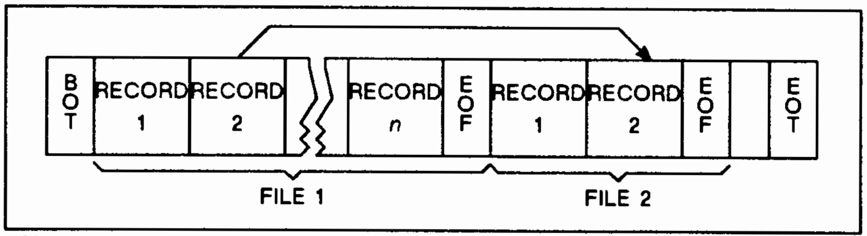
The record position is set to the next record in file 1 by incrementing RECD'POSITION with the statement:

```
RECD'POSITION:=RECD'POSITION+1;
```

and spaces ahead to that record with the statement:

```
FSPACE(MT,RECD'POSITION);
```

and the copy loop is repeated. After the copy loop is repeated, the tape is as follows:



LG200016_041

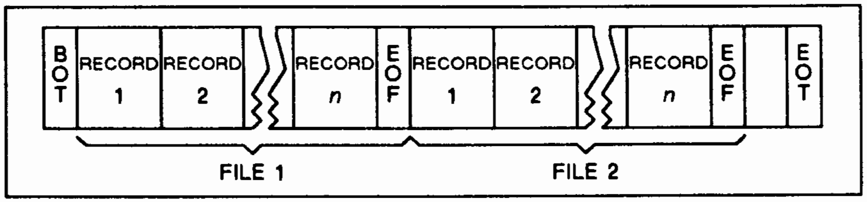
Note that the reverse tape motion after a write creates an EOF mark (see end of FILE 2).

The copy loop is repeated until the end of FILE 1 is reached, at which point program control is transferred to the statement label DONE. The tape then is rewound with the statement:

```
FCONTROL(MT,5,DUMMY);
```

and closed with the same disposition (old permanent) as before.

The format of the tape at the end of the copy operation is:



LG200016_042

Labeled Tapes

MPE provides a means whereby you can read and write labels on magnetic tape files. Labeled tapes are intended to provide for:

- A permanent identification for tape reels, or volumes.
- Files which extend over more than one volume.
- More than one file on a volume.
- Retrieval of files by file name.
- Additional security, to protect against invalid erasure or access to files.

When each tape volume is first written, it is assigned a unique identifier consisting of up to six alphanumeric characters. This identifier is the volume name. It is often strictly numeric, and volumes in an installation's library can be sorted by this number for storage.

A collection of volumes containing one or a related group of files is called a volume set. The volume name of the first reel in the set is taken as the volume set name.

Each file on a labeled tape has a header label or labels which describe the name of the file, the sequential position of the file on the Volume, and the sequential number of the volume in the volume set. Optionally, the header label may also contain the record and block size, a file lockword, and whether the file is ASCII or binary.

When opening a labeled tape file to be read, you must specify the volume set name. This may appear either in a file equation (`;LABEL` = parameter), or in the *formsmsg* parameter of `FOPEN`; if it appears in neither place, the Console Operator will be prompted for the volume set name. You may also specify whether to seek a particular file name within the volume set, or simply to access the next sequential file. If the file name you specify does not exist, an End of Volume Set error (`FSERR 123`) will be returned by `FOPEN`.

When opening a labeled tape file to be written, you specify the volume set name for reading. You may declare that a specified named file, or the next sequential file, is to be written, or that a file is to be added to the end of the volume set. An End of Volume Set error will be returned if the specified file is not found. Of course, if there are other files following the file to be written, their contents will be lost.

You may close a file without closing the volume set containing it. This means a subsequent `FOPEN` specifying the same volume set name will be able to access a file on the currently mounted volume of the volume set without operator intervention. The volume thus accessed need not be the first one in the volume set.

There are two standard formats for labels in common use: IBM and ANSI. Except that IBM labels are written in EBCDIC, the differences between them are minor. The MPE Tape Labels system can read and write labeled tapes that conform to the ANSI standard, and read tapes that conform to the IBM standard. Only ANSI standard tapes support file lockwords.

According to ANSII Standards, blocks within a file are padded out to the desired length as necessary with the circumflex character. MPE uses two types of pads, blanks for ASCII type files and nulls (0s) for binary type files. Users should ignore record

Writing a Tape Label

The MPE `FILE` command or `FOPEN` intrinsic is used to write ANSI-standard tape labels; MPE will not write IBM-standard tape labels. See the *MPE V Commands Reference Manual* (32033-90006), for a discussion of writing tape labels with the `FILE` command.

The program shown in Figure 9-3 opens a magnetic tape file and writes a label on the file.

```
$CONTROL USLINIT
BEGIN
      BYTE ARRAY FILID1(0:8):="";
      BYTE ARRAY FILID2(0:8):="NEWTAPE1 ";
      BYTE ARRAY LABELID(0:25):="FIL099,ANS,12/31/81,NEXT;";
      BYTE ARRAY DEV(0:4):="TAPE ";

ARRAY MSGBUF(0:35);
ARRAY INBUF(0:39);
ARRAY FIL'ID1(*)=FILID1;
ARRAY USERLABL(0:79);

INTEGER FN01,FN02,LGTH;

INTRINSIC FOPEN,FCLOSE,PRINT'FILE'INFO,QUIT,PRINT,
   READ,FWRITELABEL,FREAD,FWRITE;

PROCEDURE FILERROR(FILENO,QUITNO);
     VALUE QUITNO;
     INTEGER FILENO,QUITNO;
     BEGIN
   PRINT'FILE'INFO(FILENO);
   QUIT(QUITNO);
     END;

<<END OF DECLARATIONS>>

MOVE MSGBUF:="NAME OF INPUT FILE?";
PRINT (MSGBUF,-19,0);
READ(FIL'ID1,-8); <<READ NAME OF INPUT FILE>>

FN01:=FOPEN(FILID1,1,5); <<OPEN OLD DISC FILE>>
IF < THEN <<CHECK FOR ERROR>>
     BEGIN
   MOVE MSGBUF:="CAN'T OPEN DISC FILE";
   PRINT(MSGBUF,-20,0);
   FILERROR(FN01,1);
END;

FN02:=FOPEN(FILID2,%1004,5,,DIV,LABELID); <<OPEN NEW LABELED TAPE FILE>>
IF < THEN <<CHECK FOR ERROR>>
      BEGIN
    MOVE MSGBUF:="CAN'T OPEN TAPE FILE";
    PRINT(MSGBUF,-20,0);
    FILERROR(FN02,2);
   END;
```

Figure 9-3. Writing to a Tape File (1 of 2)

```
      MOVE USERLABL:="";
      MOVE USERLABL : =USERLABL(0),(40);
      MOVE USERLABL:="UHL1 USER HEADER LABEL NO. 1";
      FWRITELABEL(FNO2,USERLABL,40,0); <<WRITE USER HEADER LABEL>>
      IF < > THEN FILERROR(FNO2,3);
            <<CHECK FOR ERROR>>

 READ'WRITE'LOOP:

      LGTH:=FREAD(FNO1,INBUF,40); <<READ RECORD FROM DISC FILE>>
      IF < THEN <<CHECK FOR ERROR>>
     BEGIN
     MOVE MSGBUF:="CAN'T READ DISC FILE";
     PRINT(MSGBUF,-20,0);
     FILERROR(FNO1,4);
      END;
 IF > THEN GO CLOSE; <<CHECK FOR END-OF-FILE>>


 FWRITE(FNO2,INBUF,LGTH,0); <<WRITE RECORD TO LABELED TAPE FILE>>
IF <> THEN <<CHECK FOR ERROR>>

      BEGIN
     MOVE MSGBUF:="CAN'T WRITE TO TAPE FILE";
     PRINT(MSGBUF,-24,0);
     FILERROR(FNO2,5);
      END;

   CLOSE:

 FCLOSE(FNO1,0,0); <<CLOSE DISC FILE>>
 IF < THEN <<CHECK FOR ERROR>>
      BEGIN
     MOVE MSGBUF:="CAN'T CLOSE DISC FILE";
     PRINT(MSGBUF,-21,0);
     FILERROR(FNO1,6);
      END;


 FCLOSE(FNO2,1,0); <<CLOSE, REWIND, AND UNLOAD TAPE FILE>>
 IF < THEN <<CHECK FOR ERROR>>
 BEGIN
      MOVE MSGBUF:="CAN'T CLOSE TAPE FILE";
      PRINT(MSGBUF,-21,0);
      FILERROR(FNO2,7);
 END;
    END.
```

Figure 9-3. Writing to a Tape File (2 of 2)

The statement:

```
    BYTE ARRAY LABELID(0:25):="FIL099,ANS,12/31/81,NEXT;";
```

declares a byte array of 26 bytes and initializes it to:

```
    .FIL099,ANS,12/31/81,NEXT;
```

which specifies that ANSI-standard labels will be used. Note that the tape label statement begins with a period and ends with a semicolon. This is necessary to distinguish the tape label statement from a forms message (which is another use for the same **FOPEN** parameter). The **LABELID** byte array will be used in the **FOPEN** Intrinsic call to specify a file label as follows:

Volume Identification: FIL099

Label Type: **ANS** (ANSI)

Expiration Date: **12/31/81**. This is the date after which the file can be overwritten. If you attempt to overwrite the file before this date, MPE will send a message to the Console Operator asking for confirmation that such is really desired. This affords an extra measure of protection against inadvertently destroying a tape by overwriting when a WRITE RING is left on the tape by mistake.

Sequence: **NEXT**, Signifies that the file is to be positioned at the next file on the tape.

The statement:

```
    FN02:=FOPEN(FILID2,%1004,5,,DEV,LABELID,1);
```

opens a new tape file and writes the tape label as specified by LABELID.

Opening a Labeled Magnetic Tape File

Figure 9-4 shows a program that opens a labeled magnetic tape file and a disk file, reads the contents of the tape file and writes the records to the disk file, closes the tape file, and finally closes the disk file as a permanent file.

```
    $CONTROL USLINIT
    BEGIN
   BYTE ARRAY FILID1(0:8):="TAPEFILE ";
   BYTE ARRAY FILID2(0:8):=" ";
   BYTE ARRAY LABELID(0:25):=".FIL001,ANS,12/31/81,,,";
   BYTE ARRAY DEV(0:4):="TAPE ";

   ARRAY MSGBUF(0:35);
   ARRAY INBUF(0:39);
   ARRAY FIL'ID2(*)=FILID2;

   INTEGER FN01,FN02,LGTH;

   INTRINSIC FOPEN,FCLOSE,FREAD,FWRITE,READ,PRINT,PRINTFILEINFO,
   QUIT,CAUSEBREAK,FREADLABEL;

   PROCEDURE FILERROR(FILENO,QUITNO);
VALUE QUITNO;
INTEGER FILENO,QUITNO;
BEGIN
    PRINT'FILE'INFO(FILENO);
    QUIT(QUITNO);
END;

  << END OF DECLARATIONS >>

   MOVE MSGBUF:="NAME OF NEW DISC FILE TO BE CREATED?";
   PRINT(MSGBUF,-8,0);
   READ(FIL'ID2,4); <<READ NAME OF NEW DIS FILE>>

   FN01:=FOPEN(FILEID1,%1001,,,DEV,LABELID);<<OPEN LABELED TAPE FILE>>
    IF < THEN <<CHECK FOR ERROR>>
    BEGIN
MOVE MSGBUF:="CAN'T OPEN TAPE FILE";
PRINT(MSGBUF,-20,0);
FILERROR(FN01,1);
  END;

   FN02:=FOPEN(FILID2,4,5); <<OPEN NEW DISC FILE>>
   IF < THEN <<CHECK FOR ERROR>>
  BEGIN
MOVE MSGBUF:="CAN'T OPEN DISC FILE";
PRINT(MSGBUF,-20,0);
FILERROR(FN02,2);
   END;

    FREADLABEL (FN01,INBUF,40); <<READ FOR USER LABEL>>
    IF <> THEN FILERROR (FN01,3); <<CHECK FOR ERROR>>
```

Figure 9-4. Opening a Label Magnetic Tape File (1 of 2)

```
   PRINT(INBUF,40,0);

    READ’WRITE’LOOP;

   LGTH:=FREAD(FN01,INBUF,40); <<READ RECORD FROM TAPE FILE>>
   IF < THEN <<CHECK FOR ERROR>>
 BEGIN
      MOVE MSGBUF:="CAN’T READ TAPE FILE";
      PRINT(MSGBUF,-20,0);
      FILERROR(FN001,4);
 END ;
   IF > THEN GO CLOSE1; <<CHECK FOR END-OF-FILE>>

   FWRITE(FN02,INBUF,LGTH,0) <<WRITE RECORD TO DISC FILE>>
   IF <> THEN <<CHECK FOR ERROR>>
  BEGIN
 MOVE MSGBUF:="CAN’T WRITE TO DISC FILE";
 PRINT(MSGBUF,-24,0);
 FILERROR(FN02,5);
   END;

    GOTO READ’WRITE’LOOP;

    CLOSE1:

   FCLOSE(FN01,1,0); <<CLOSE, REWIND, AND UNLOAD TAPE FILE>>
   IF < THEN <<CHECK FOR ERROR>>
 BEGIN
MOVE MSGBUF:="CAN’T CLOSE TAPE FILE" ;
PRINT(MSGBUF,-21,0);
FILERROR(FN001,6);
 END;

    CLOSE2:

   FCLOSE(FN02,1,0); <<CLOSE DISC FILE AS PERMANENT FILE>>
   IF < THEN <<CHECK FOR ERROR>>
 BEGIN
      MOVE MSGBUF:="CAN’T CLOSE DISC FILE";
      PRINT(MSGBUF,-21,0);
      MOVE MSGBUF:="CHECK FOR DUPLICATE NAME";
      PRINT(MSGBUF,-24,0);
      MOVE MSGBUF:="FIX, THEN TYPE ’RESUME";
      PRINT(MSGBUF,-23,0);
      GOTO CLOSE2; <<TRY AGAIN>>
   END;
    END.
```

Figure 9-4. Opening a Labeled Magnetic Tape File (2 of 2)

The statement:

```
FN01:=FOPEN(FILID1,%1004,,,DEV,LABELID);
```

calls `FOPEN` to open the labeled magnetic tape file. The parameters specified are:

formaldesignator `TAPEFILE`, stored in the byte array `FILID1`.

foptions %1005, for which the bit pattern is:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 Bits

0 0 0 0 0 0 1 0 0 0  0  0  0  1  0  1 Binary

 1       0       0           5     Octal
```

The above bit pattern specifies the following file options:

Domain: Old, permanent file, system file domain.
Bits(14:2) =01.
File Designator: Actual file designator same as formal file designator.
Bits(10:3) = 000. (Default)
Labeled Tape: Labeled. Bit(6:1) = 1.

Note

The following are specified but will be overridden by the file attributes on the tape if it is an HP-labeled tape:

ASCII/Binary: ASCII. Bit (13:1)=1. Record Format: Fixed length. Bits (8:2)=0. (Default) Carriage Control: No carriage control. Bit (7:1)=0. (Default)

aoptions Omitted. Default of read access, buffered, no multi will be used.

recsize Omitted. The record size specified by the tape label will be used.

device `TAPE`, contained in the byte array `DEV`.

tape label Contained in the byte array `LABELID` (*formmsg* parameter). `LABELID` is initialized with the value

```
.FIL001,ANS,12/31/81,,;
```

(See statement number 5 in Figure 9-4). Note that the tape label begins with a period and ends with a semicolon. This is necessary to distinguish the tape label from a forms message (another use for this parameter).

When the FOPEN intrinsic call executes, MPE sends a message to the system console, requesting the Console Operator to mount the tape labeled FIL001, if it is not already mounted.

The statement:

```
FNO2:=FOPEN(FILID2,4,5);
```

opens a new disk file.

The program then reads records from the tape file with the statement:

```
LGTH:=FREAD(FNO1,INBUF,40);
```

and writes these records to the disk file with the statement:

```
FWRITE(FNO2,INBUF,LGTH,0);
```

When all records in the tape file have been read, both files are closed. The disk file is saved as a permanent file.

Reading a Labeled Magnetic Tape File

Once a labeled tape file has been opened, the FREAD intrinsic may be used in the same manner as on an unlabeled tape file. The system uses the blocksize, recordsize and file format on the tape label. You can call FGETINFO or FFILEINFO to get these values.

The program shown in Figure 9-4 reads a labeled magnetic tape file in sequential order.

The labeled tape file is opened with the statement:

```
FNO1:=FOPEN(FILID1,%1005,5,,DEV,LABELID);
```

The file label is contained in the byte array LABELID.

The block of statements:

```
READ'WRITE'LOOP:
⋮
GOTO READ'WRITE'LOOP;
```

forms a read/write loop. Records are read from the tape file in sequential order with the statement:

```
LGTH:=FREAD(FNO1,INBUF,40);
```

and written to a disk file with the statement:

```
FWRITE(FNO2,INBUF,LGTH,0);
```

Writing to a Labeled Magnetic Tape File

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 volume (reel of tape) to be mounted.

The program shown in Figure 9-3 opens an existing disk file and a new labeled tape file, reads records from the disk file and writes these records to the tape file. If an attempt is made to write records on the tape beyond the EOT marker, MPE will write EOV1 and EOV2 labels on the tape and request the Console Operator to mount another reel of tape.

The statement:

```
FWRITE (FNO2,INBUF,LGTH,0);
```

writes the contents of array INBUF onto the tape file signified by FNO2. The LGTH parameter specifies the number of words to be written.

Writing a User-Defined File Label on a Labeled Tape File

User-defined labels are used to further identify files and may be used in addition to the ANSI-standard labels. Note that user-defined labels may not be written on unlabeled magnetic tape files.

User-defined labels are written on files with the FWRITELABEL intrinsic instead of the 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 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.

The program in Figure 9-3 opens a new tape file and writes an ANSI-standard label on it, then writes a user-defined header label with the FWRITELABEL intrinsic.

The statement:

```
FNO2:=FOPEN(FILID2,%1004,5,,DEV,LABELID,1);
```

opens a new tape file named NEWTAPE1 (the name is contained in byte array FILID2) and writes an ANSI-standard label (specified by byte array LABELID) to the file.

The statements:

```
MOVE USERLABL:=" ";
MOVE USERLABL:=USERLABL(0),(40);
```

fill the array USERLABL with 80 ASCII blanks (40 words), and the statement:

```
MOVE USERLABL:="UHL1 USER HEADER LABEL NO. 1";
```

moves the desired user label into the first 35 bytes of the array, replacing the blanks.

The statement:

```
FWRITELABEL(FNO2,USERLABEL,40,0);
```

writes all 80 characters into the file as a user-defined header label.

Note that in order to write a user-defined header label, the FWRITELABEL intrinsic must be called before the first FWRITE to the file. MPE will, however, write user-defined trailer labels if FWRITELABEL is called after the first FWRITE.

Reading a User-Defined File Label on a Labeled Tape File

The FREADLABEL intrinsic is used to read a user-defined label on a labeled magnetic tape 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 to skip past any unread user-defined header labels.

In Figure 9-4, the statement:

```
FREADLABEL(FNO1,INBUF,40);
```

reads a user-defined header label. The parameters specified are:

FNO2 The file number as returned by the FOPEN intrinsic.

INBUF An array to which the label is transferred.

'@' Specifies the number of words to be read.

Storing Files Offline

You can obtain a backup copy of a particular user disk file or set of files by storing it offline onto magnetic tape or serial disk with the STORE command. Use the RESTORE command to restore files from magnetic tape. Refer to *MPE V/E Storing and Restoring Files* (32033-90133) for instructions.

Section Divider

A. File System Reference

A

File System Reference

File System Reference

Records always begin and end on word boundaries (odd byte length records padded out to a word boundary).

Record Formats

The three record formats available with the File System are fixed format, variable format, and undefined length format.

Fixed Length

All records are fixed length; blocks contain a fixed number of records.

Record length and blocking factor are known to file system.

Records consist of data only.

Variable Length

Record length varies; blocks contain a variable number of records.

File attribute defined to file system is Maximum Record Length = (Record Length + 1) x Blocking Factor. This is the largest data record the file can accommodate.

Block length is Maximum Record Length plus 1 word.

Each record consists of data plus a field containing length of that record in bytes (field is 1 word long).

Each block contains an end-of-block indicator (1 word long).

Undefined Length

Block length is set to record length defined to file system.

Blocking factor is assumed to be 1.

Blocking and deblocking of records is the user's responsibility.

Buffering

The syntax for buffering with the FILE command is:

```
FILE ... [;BUF [=numbuffers] ]
         [;NOBUF ]
```

Block length is buffer size.

Default is two buffers.

May be overridden at run time with FILE command.

NOBUF specifies no buffers allocated for this file. Blocks transferred directly onto user's stack.

File system performs no blocking/deblocking.

Parameters Common to FILE and BUILD Commands

Parameters common to both the FILE and BUILD Commands are shown in the syntax:

```
                                          [F]
[;REC=[recsize] [,[blockfactor] [,[U] [,BINARY]]]]
                                          [V] [,ASCII]
```

recsize + for words, – for bytes.

BINARY Pad longer records or new disc extents with binary zeroes (%0).

ASCII Pad longer records or new disc extents with spaces (%40).

[;DISC=[*numrec*] [,[*numextents*][,*initalloc*]]]

numrec Maximum number of records to allow in file. Default = 1023.

numextents 1 through 32; default is 8.

initalloc Number of extents initially allocated. Default = 1.

[;CODE] User codes are 0 through 1023. Negative codes accessible only in Privileged Mode. 1024+ or mnemonic are system defined and shown in Table A-1.

Table A-1. System-Defined Mnemonic Codes

Integer	Mnemonic	Meaning
1024	USL	User Subprogram Library
1025	BASD	BASIC Data
1026	BASP	BASIC Program
1027	BASFP	BASIC Fast Program
1028	RL	Relocatable library
1029	PROG	Program file
1031	SL	Segmented Library
1035	VFORM	View Form file
1036	VFAST	View Fast Forms file
1037	VREF	View Reformat file
1040	XLSAV	Cross Loader ASCII file (SAVE)
1041	XLBIN	Cross Loader Relocated Binary file
1042	XLDXP	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 Workfile
1059	TDPXQ	TDP Workfile (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 logfile
1100	WDOC	HPWORD Document
1101	WDICT	HPWORD Hyphenation dictionary
1102	WCONF	HPWORD Configuration file
1103	W2601	HPWORD Attended Printer Environment
1110	PCELL	IFS/3000 Character Cell file
1111	PFORM	IFS/3000 Form file
1112	PENV	IFS/3000 Environment file
1113	PCCMP	IFS/3000 Compiled Character Cell file
1114	RASTR	Graphics Image in RASTR Format
1130	OPTLF	OPT/3000 logfile
1131	TEPES	TEPE/3000 Script file
1132	TEPEL	TEPE/3000 logfile
1133	SAMPL	APS/3000 logfile
1139	MPEDL	MPEDCP/DRP logfile
1140	TSR	HPToolset Root file
1141	TSD	HPToolset Data file

Table A-1. System-Defined Mnemonic Codes (continued)

Integer	Mnemonic	Meaning
1145	DRAW	Drawing file for HPDRAW
1146	FIG	Figure File for HPD
1147	FONT	Reserved
1148	COLOR	Reserved
1149	D48	Reserved
1152	SLATE	Compressed SLATE file
1153	SLATW	Expanded SLATE workfile
1156	DSTOR	Store file for RAPID/3000 Utility DICTDBU
1157	TCODE	Code file for Transact/3000 Compiler
1158	RCDOE	Code file for Report/3000 Compiler
1159	ICODE	Code file for Inform/3000 Compiler
1166	MDIST	HPDESK Distribution list
1167	MTEST	HPDESK Test
1168	MARPA	ARPA Message file
1169	MARPD	ARPA Distribution List
1170	MCMND	HPDESK Abbreviated Commands file
1171	MFRT	Reserved
1172		Reserved
1173	MEFT	Reserved
1174	MCRPT	Reserved
1175	MSERL	Reserved
1176	UCSF	Reserved
1177	TTYPE	Term Type file
1178	TVFC	Term Vertical Format Control file
1192	NCONF	Network Configuration file
1193	NTRAC	Network Trace file
1194	NLOG	Network logfile
1195	MIDAS	Reserved
1211	ANODE	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	DIFfile
1228	LANGD	Language Definition file
1229	CHARD	Character Set Definition file
1230	MGCAT	Formatted Application Message Catalog
1236	BMAP	Reserved
1242	BDATA	Basic Data file

Table A-1. System-Defined Mnemonic Codes (continued)

Integer	Mnemonic	Meaning
1243	BFORM	Basic Field Order File for VPLUS
1244	BSAVE	Basic Saved Program File
1245	BCNFG	Configuration File for Default Option Basic Program
1258	PFSTA	Pathflow STATIC file
1259	PFDYN	Pathflow DYNAMIC file
1270	RTDCA	Revisable Form DCA Document (DCA=Document Content Architecture)
1271	FFDCA	Final Form DCA Document (DCA=Document Content Architecture)
1272	DIU	Document Interchange Unit file
1273	PDOC	HPWORD/150 Document
1401	CWPTX	Reserved
1421	MAP	HPMAP/3000 Map Specification file
1422	TTX	Reserved
3333		Reserved

Default is the unreserved file code of 0.

Using 1090 (LOG) as your designated *filecode* may not yield the number or records 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.

```
[;CCTL]
[;NOCCTL]
```

CCTL — An additional character is added to the beginning of each record containing carriage control information, in addition to record length. Valid for ASCII files only.

NOCCTL — No additional character reserved for carriage control. (Default = NOCCTL.)

[;TEMP] — The BUILD command can only create a file in the Permanent or Temporary domain. Default is Permanent.

Referencing Disc File Domains

The syntax for referencing disc file domains is:

```
           [,NEW ]     [;DEL ]
FILE ...  [,OLD ]     [;SAVE]
           [,OLDTEMP] [;TEMP]
```

NEW — Create a disc file in the NEW domain.

OLD — Find a disc file that already exists in the OLD (PERMANENT) domain.

`OLDTEMP`	Find a disc file that already exists in the TEMPORARY domain.
Default	Search TEMPORARY domain then PERMANENT domain.
`DEL`	Delete file upon close.
`SAVE`	Move this file to PERMANENT domain upon close.
`TEMP`	Make this NEW file TEMPORARY upon close.
	Default Upon close, file is left in the domain in which it was found when opened. (New files are deleted.)

FILE Back-Reference

The syntax used with the `FILE` command to back-reference files is:

`FILE` *formaldesignator1* = **formaldesignator2*

Here *formaldesignator1* takes on all the same attributes as *formaldesignator2* from a previous or subsequent `FILE` command.

Controlling Simultaneous Access to Disc Files

The syntax for controlling simultaneous access to disc files is:

```
           [;EXC ]
FILE ...   [;SEMI]
           [;SHR ]
```

`EXC`	Exclusive access. No other users will be allowed to access this file while you have it open. You will not be allowed EXC access if someone is already using the file.
`SEMI`	Exclusive Allowing Read. Other users may open file but only for read-only access (`ACC=IN`). You will be granted this access only if no one else is using this file or It is opened for read-only access.
`SHR`	Shared access. Allow concurrent use by other users. You will not be granted access to the file if someone has it opened with `EXC` access.

Specifying Access

The syntax to specify access is:

```
                 IN
                 OUT
                 INOUT
FILE ... [:ACC = OUTKEEP]
                 APPEND
                 UPDATE
```

`IN`	Read only.
`OUT`	Write only. Original contents of file overwritten. File cannot be read.

INOUT write — Any operation except update is allowed. (You can still read then the same record.)

OUTKEEP — Write-only access. Original contents kept and you are allowed to write both before and after end-of-file. File cannot be read.

APPEND — Records may be written only beyond end-of-file. File cannot be read nor can you write into the original part of the file.

UPDATE — Update access. All operations may be performed on file.

Default — IN access for devices that can perform input; otherwise OUT for output-only devices.

Specialized Parameters of FILE

The specialized parameters of the FILE command are:

MULTI: — Multi-user access; requires Process Handling capability.

MR: — Allows multiple block access.

NOWAIT: — Do not wait for I/O completion; requires Privileged Mode capability.

User Types

User codes, for security specification, and their definitions are:

ANY — Any User. This category covers any user defined in the system, and includes all categories defined below.

AL — Account Librarian User. User with Account Librarian capability, who can manage certain files within his account that may or may not all belong to one group.

GL — Group Librarian User. User with Group Librarian capability, who can manage certain files within his home group.

CR — Creating User. The user who created this file.

GU — Group User. Any user allowed to access this group as his log-on or home group, including all GL users applicable to this group.

AC — Account Member. Any user authorized access to the system under this account; this includes all AL, GU, GL, and CR users under this account.

Figure A-1 shows the *FOPTIONs* available with the FOPEN intrinsic.

BITS	(0:2)	(2:3)		(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	Reserved	File Type		Disallow :FILE	MPE Tape Labels	Carriage Control	Record Format	Default Designator	ASCII/ Binary	Domain
MEANING		00	0=STD	0=Allow :FILE	0=NON LABELED TAPE	0=NOCCTL	00=Fixed	000=Filename	0=Binary	00=New File
		00	1=KSAM	1=No :FILE	1=LABELED TAPE	1= CCTL	01=Variable	001=$STDLST	1=ASCII	01=Old Permanent File
		01	0=RIO				10=Undefined	010=$NEWPASS		
		10	0=CIR					011=$OLDPASS		10= Old Temporary File
		11	0= MSG					100=$STDIN		
								101=$STDINX		11=Old Perm. or Temp. File
								110=$NULL		

Figure A-1. FOPTIONs for Use with FOPEN

Figure A-2 shows the *AOPTIONs* available with the FOPEN intrinsic.

BITS	(0:3)	(3:1)	(4:1)	(5:2)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)	
FIELD	Reserved	File Copy	No-Wait I/O	Multi Access	Inhibit Buffering	Exclusive Access	Dynamic Locking	Multi-record Access	Access Type	
MEANING		0=Access in file's native mode	1=No Wait	00=Non-multi-access	0=BUF	00=Default	0=No FLOCK Allowed	0=No Multi-Record	0	000=Read Only
		1=Access as standard sequential file	2=Non No-Wait	01=Only intra=job multi-access	1=NOBUF	01=Exclusive	1=FLOCK Allowed	1=Multi-Record	0	001=Write only
				10=Inter-job multi-access allowed		10=Exclusive Access Read			0	010=Write (save) only
						11=Share			0	011=Append only
									0	100=Read/write
									0	101=Update
									0	110=Execute

Figure A-2. AOPTIONS for Use with FOPEN

MPE defaults and device-dependent restrictions are shown in Figure A-3.

```
INPUT ONLY DEVICES (SERIAL)

   Card Reader/Paper Tape Reader

     No carriage control
     Undefined-length records if card reader, ASCII only (can only read ASCII
       cards using FCONTROL)
     Blockfactor = 1
     Domain = 1 (OLD permanent)
     If not ASCII, then NOBUF
     If access type = 1,2,3, then access violation results

INPUT/OUTPUT DEVICES (PARALLEL)

   Terminals

     ASCII
     NOBUF
     Undefined=length records
     Blockfactor = 1

INPUT/OUTUT DEVICES (SERIAL)

   Magnetic Tape Drive
   Serial Disc Drive
     No restriction

OUTPUT ONLY (SERIAL)

   Line Printer/Card Punch/Paper Tape Punch/Plotter

     If Paper Tape punch, ASCII only
     Undefined-length records
     Blockfactor = 1
     Domain = NEW
     Access Type = 1, write only (if read only specified, access violation results)

Laser Printer

   Initially and always spooled
   Write only access
   All other restrictions same as for line printer

UNDEFINED (COMMON CHECKING)

   If carriage control specified and not ASCII, access violation results
```

Figure A-3. MPE Defaults and Device-Dependent Restrictions

The Relative I/O Block Format is shown in Figure A-4.

```
Item
#23       Logical Record 0
                  .
                  .                  Item
                  .                  #22
                  .                            Item
                  .                            #21

          Logical Record F-1
Item                                 Item
#24       Active Record Table        #25

FFILEINFO Item Numbers

  Item 21 - Physical Block Size
       22 - Data Block Size
       23 - Offset to Data in Blocks
       24 - Offset to Active Record Table within the block
       25 - Size of Active Record Table

Active Record Table
                                               record 0 (block-relative)
                                               record 15
       | | | | | | | | | | | | | | |           word 0


       | | | | | | | | | | | | | | |           word A-1
                           1 1 1 1 1 1
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

A = active-record table size in words= F
                                       -
                                       16

F = blocking factor (number of records per block)

R = index of desired record, modulo F

W = index of word for desired record = R/16

P = index of bit for desired record = R mod 16

bit = 0 :inactive record
    = 1 :active record
```

Figure A-4. Relative I/O Block Format

Section Divider

B. Status Information

B

Status Information

You can use certain file system intrinsics to obtain information on the status of your disc files. Information is available about actual file characteristics, current file information, and error information:

- *Actual file characteristics* include the physical and operational features of your file. Such characteristics are defined by a combination of FOPEN parameters, FILE commands, file label contents, and file system defaults.
- *Current file information* includes details on the current status of your file, such as the placement of the end-of-file indicator, the location of the record pointer, and the logical and physical record transfer count.
- *Error information* lists the last error for your file or the last FOPEN error.

Obtaining Status Information

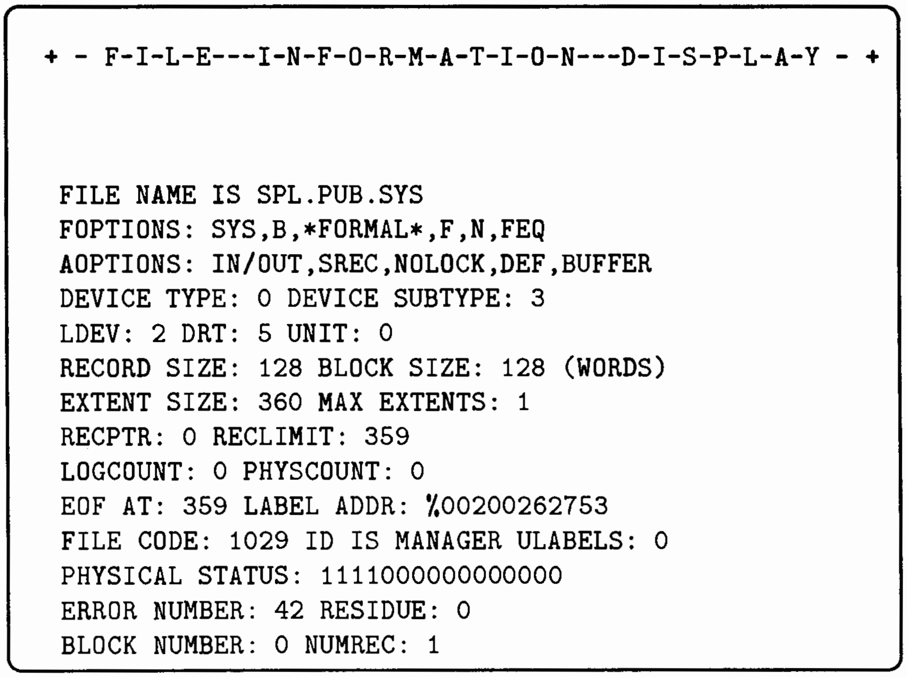
The same status information may be obtained via different intrinsic calls. The FGETINFO and FFILEINFO intrinsics will return actual file characteristics and current file information, the FCHECK intrinsic will return error information, and the PRINTFILEINFO intrinsic will list details of all three categories. PRINTFILEINFO will format information and output it to the list device for your job/session; FGETINFO, FFILEINFO, and FCHECK will return unformatted information directly to your calling program.

PRINTFILEINFO

The PRINTFILEINFO intrinsic requires the file number returned by an FOPEN call; in the case of an FOPEN failure, give zero as the file number. Output will be printed to your job/session list device in one of two formats; a full file information display for open files (see Figure B-1) or a short file information display for files that are not open (see Figure B-2).

Note

These formats are sometimes referred to as *tombstones.* This may give the impression that the executing process aborts, but this is not so; a file information display is simply a listing of status.



```
+ - 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 SPL.PUB.SYS
 FOPTIONS: SYS,B,*FORMAL*,F,N,FEQ
 AOPTIONS: IN/OUT,SREC,NOLOCK,DEF,BUFFER
 DEVICE TYPE: 0 DEVICE SUBTYPE: 3
 LDEV: 2 DRT: 5 UNIT: 0
 RECORD SIZE: 128 BLOCK SIZE: 128 (WORDS)
 EXTENT SIZE: 360 MAX EXTENTS: 1
 RECPTR: 0 RECLIMIT: 359
 LOGCOUNT: 0 PHYSCOUNT: 0
 EOF AT: 359 LABEL ADDR: %00200262753
 FILE CODE: 1029 ID IS MANAGER ULABELS: 0
 PHYSICAL STATUS: 1111000000000000
 ERROR NUMBER: 42 RESIDUE: 0
 BLOCK NUMBER: 0 NUMREC: 1
```

Figure B-1. File Information Display - Full

File is open:

- File number represents a currently open file.
- Error indicates last error on file.
- Full display condensed when file is not open.

```
 + - F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y - +


FILE NUMBER -1 IS UNDEFINED.

ERROR NUMBER: 52 RESIDUE: 0

BLOCK NUMBER: 0 NUMREC: 0
```

Figure B-2. File Information Display - Short

File is not open:

- File number is zero or invalid.
- FOPEN failure assumed if zero file number (first line not printed) or invalid file number.
- Error is always last FOPEN error.

Sections of the full File Information Display yield different types of information, as indicated below.

Name and options:

```
    FILE NAME IS SPL.PUB.SYS
    FOPTIONS:  SYS,  B,*FORMAL*,F,N.FEQ
    AOPTIONS:  IN/OUT,SREC,NOLOCK,DEF,BUFFER
```

Device and data structure:

```
DEVICE TYPE:  0       DEVICE SUBTYPE:   3
LDEV:  2            DRT:  5                  UNIT:   0
RECORD SIZE:   128       BLOCK SIZE:   128         (WORDS)
EXTENT SIZE:   360        MAX EXTENTS:   1
```

Transfer information:

```
   RECPTR:   0            RECLIMIT:   359
   LOGCOUNT:   0            PHYSCOUNT:   0
   EOP AT:   359
```

Labels and physical status:

```
                         LABEL ADDR:   %00200262753
FILE CODE:  1029     ID IS MANAGER      ULABELS:  0
PHYSICAL STATUS:   1111000000000000
```

Error information:

```
ERROR NUMBER:   42                RESIDUE:   0
BLOCK NUMBER:   0                          NUMREC:   1
```

The fields in a file information display are described in Figure B-3 through Figure B-7.

```
FILE NAME IS SPL.PUB.SYS
FOPTIONS: SYS,B,*FORMAT*,F,N,FEQ
AOPTIONS: IN/OUT, SREC, NOLOCK, DEF, BUFFER
```

- File name: Fully qualified (name, group, account)
- *FOPTIONs:* Actual *FOPTIONs* in effect.
- *AOPTIONs:* Current *AOPTIONs* in effect.

FOPTION Keywords:

Domain	ASCII/ Binary	Default Designator	Record Format	Carriage Control	Disallow FILE
NEW	A	*FORMAL*	F	N	FEQ
SYS	B	$STDLIST	V	C	DEQ
JOB		NEWPASS	U		
ALL		$OLDPASS	?		
		$STDIN			
		$STDINX			
		$NULL			

AOPTION Keywords:

Access Type	Multi-Record	Dynamic Locking	Exclusive Access	Inhibit Buffering
INPUT	SREC	NOLOCK	DEF	BUFFER
OUTPUT	MREC	LOCK	EXC	NOBUFF
OUTKEEP			SEA*	
APPEND			SHR	
IN/OUT				
UPDATE				

*Semi-exclusive access (SEMI).
NOTE: Multi-access, NOWAIT fields not represented.

Figure B-3. Name and Options in a File Information Display

```
DEVICE TYPE: 0          DEVICE SUBTYPE: 3
LDEV: 2 DRT: 5          UNIT:0
RECORD SIZE: 128        BLOCK SIZE: 128
EXTENT SIZE: 360        MAX EXTENTS: 1
```

Device Type, Subtype,

LDEV, DRT, UNIT — Hardware Information (Set at configuration)

Record Size — Logical Record Size (Words/Bytes). For variable-length records, does not include 2 words added.

Block Size — Physical Record Size (Words/Bytes). Does not include words added for variable-length records.

Extent Size — Number of Sectors per extent.

Max Extents — Maximum allowed for file.

Figure B-4. Device and Data Structure in a File Information Display

```
RECPTR: 0          RECLIMIT: 359
LOGCOUNT: 0        PHYSCOUNT: 0
EOF AT: 359
```

RECPTR — Current record pointer (logical or physical). Points to next record to be transferred.

RECLIMIT — Maximum number of records in file.

LOGCOUNT — Number of logical record transfers to/from user stack since FOPEN.

PHYSCOUNT — Number of physical record transfers to/from file (disk) since FOPEN.

EOF AT — Current EOF pointer (one plus largest logical record number ever used to write data to the file).

NOTE: `RECPTR`, `LOGCOUNT`, `PHYSCOUNT`, `EOF` start at 0 for new file. `NOBUF`, `LOGCOUNT` = `PHYSCOUNT`; `PHYSCOUNT` updated only on completion of I/O transfer.

Figure B-5. Transfer Information in a File Information Display

```
                        LABEL ADDR:    %00200262753
   FILE CODE; 1029     ID IS MANAGER     ULABELS:  0
   PHYSICAL STATUS:   1111000000000000
```

LABEL ADDR: Sector address and ldev number for file label. First three digits for ldev: next eight digits for sector address.

FILE CODE: User or system defined (blank if zero).

ID: User name of creator.

ULABELS: Maximum number of user labels allowed.

PHYSICAL STATUS: Status of disk at time of last interrupt. (Meaningless for disk in multiprogramming environment.)

Figure B-6. Labels and Physical Status in a File Information Display

```
   ERROR NUMBER:   42                  RESIDUE:   0
   BLOCK NUMBER:   0                          NUMERIC:   1
```

ERROR NUMBER: Last error for file. 0 means EOF detected or no error occurred.

RESIDUE:

1. Number of words/bytes not transferred after error was detected.
2. In case of EOF, number of words/bytes transferred before EOF was detected.

BLOCK NUMBER: Error detected in this block.

NUMREC: Number of logical records in "error" block.

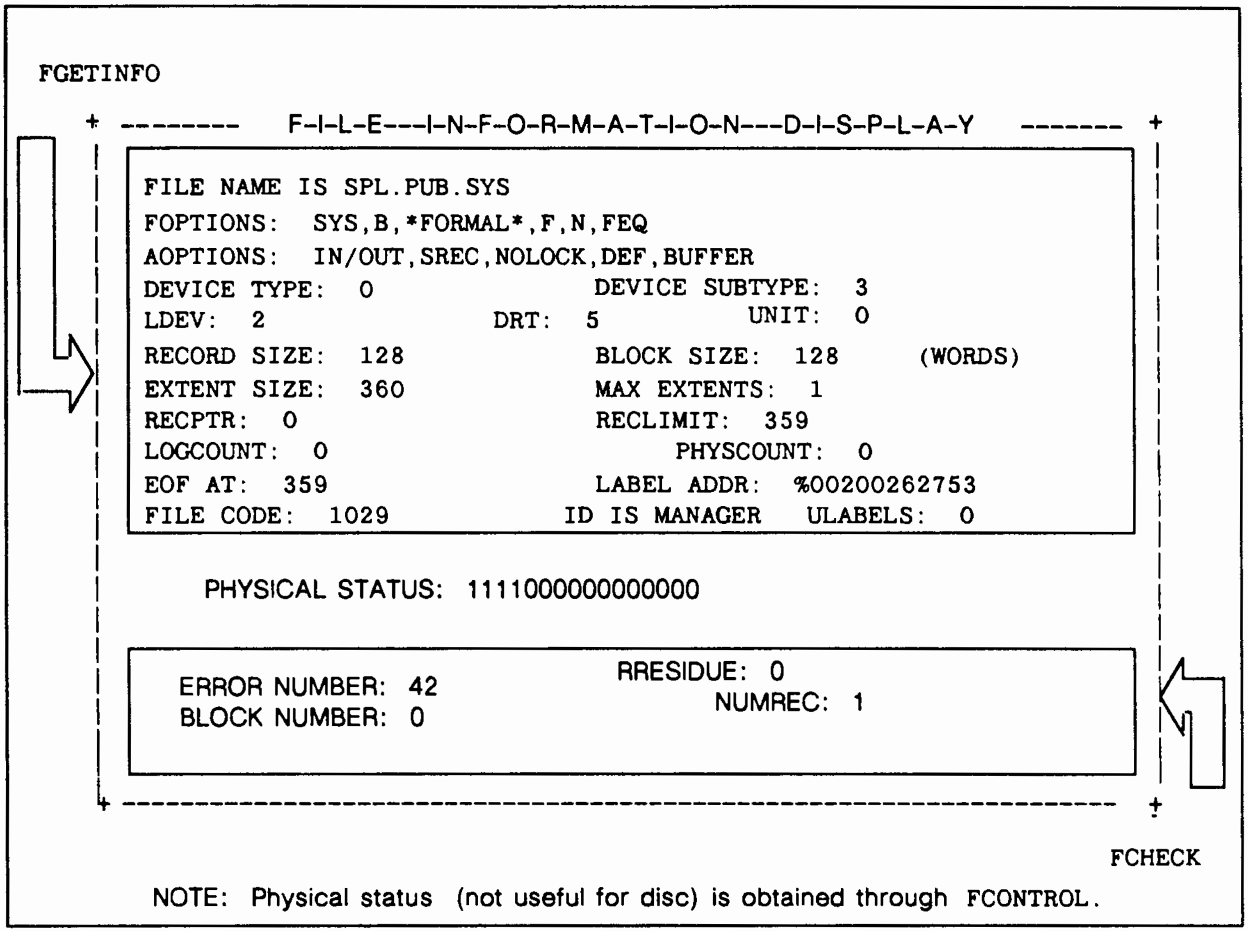
NOTE: Block number starts at 0.

Figure B-7. Error Information in a File Information Display

FGETINFO, FFILEINFO, and FCHECK

Much of the status information obtainable through the PRINTFILEINFO intrinsic can be discovered by using the FGETINFO and FCHECK intrinsics. While PRINTFILEINFO prints a file information display, FGETINFO and FCHECK return status information directly to our program through their parameters.

The information returned by FGETINFO and FCHECK that corresponds to PRINTFILEINFO information is shown in Figure B-8.



```
FGETINFO

   + --------  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 SPL.PUB.SYS                                           |
   |  FOPTIONS:   SYS,B,*FORMAL*,F,N,FEQ                                 |
   |  AOPTIONS:   IN/OUT,SREC,NOLOCK,DEF,BUFFER                          |
   |  DEVICE TYPE:  0                DEVICE SUBTYPE:  3                  |
   |  LDEV:  2            DRT:  5              UNIT:  0                  |
   |  RECORD SIZE:  128              BLOCK SIZE:  128     (WORDS)        |
   |  EXTENT SIZE:  360              MAX EXTENTS:  1                     |
   |  RECPTR:  0                     RECLIMIT:  359                      |
   |  LOGCOUNT:  0                        PHYSCOUNT:  0                  |
   |  EOF AT:  359                   LABEL ADDR:  %00200262753           |
   |  FILE CODE:  1029             ID IS MANAGER   ULABELS:  0           |
   |                                                                     |
   |      PHYSICAL STATUS:  1111000000000000                             |
   |                                                                     |
   |                                    RRESIDUE:  0                     |
   |     ERROR NUMBER:  42                     NUMREC:  1                |
   |     BLOCK NUMBER:  0                                                |
   |                                                                     |
   + ------------------------------------------------------------------- +
                                                                  FCHECK
        NOTE:  Physical status  (not useful for disc) is obtained through  FCONTROL.
```

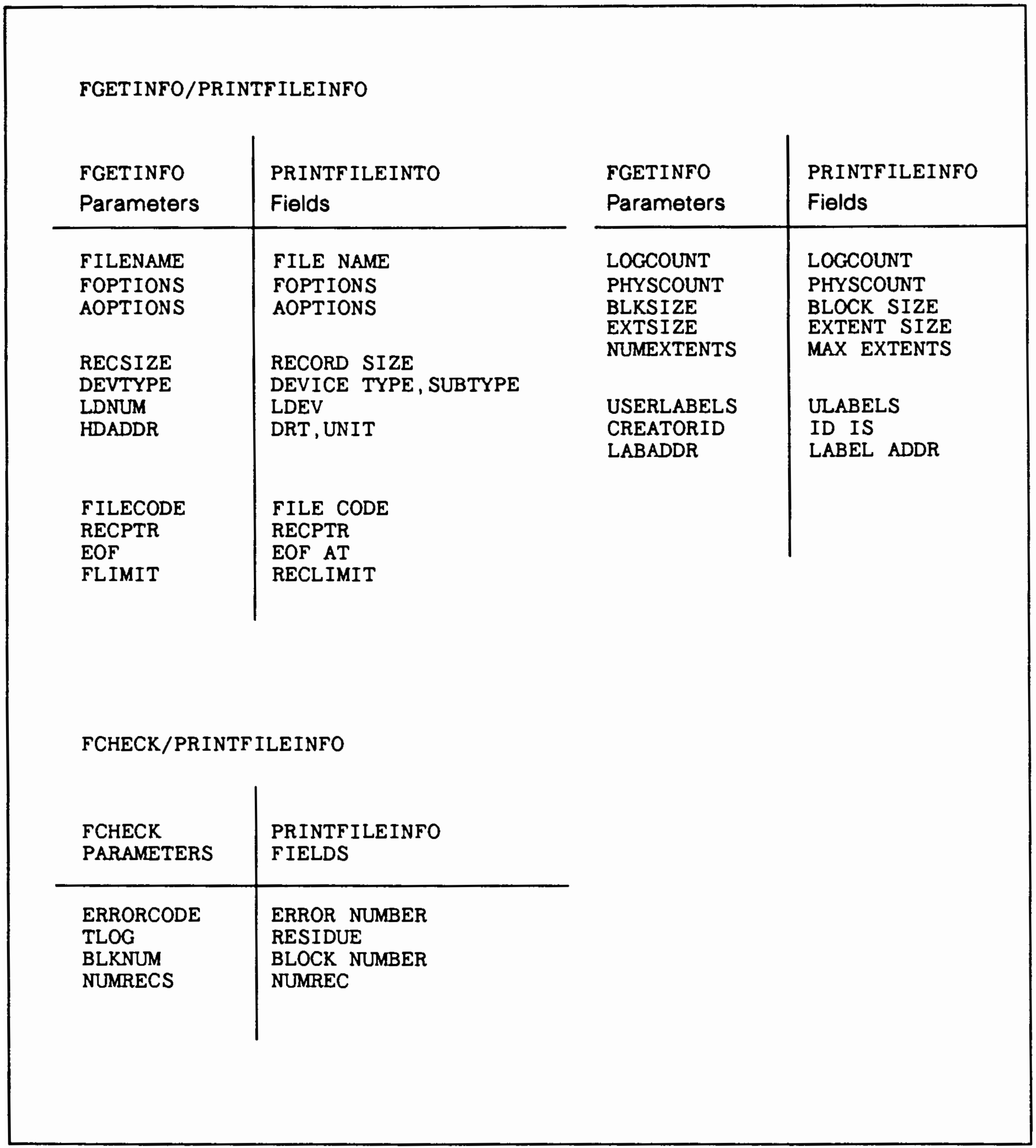
LG200016_059

Figure B-8. Information Available Through FGETINFO and FCHECK

Both FGETINFO and FCHECK require the file number returned by an FOPEN call. If you omit the file number with FCHECK, or supply a file number of zero (0), FCHECK will assume an FOPEN failure. Invalid file numbers result in error conditions for both FGETINFO and FCHECK.

Status information is returned through the parameters of FGETINFO and FCHECK. With FCHECK, the error code returned is the error which occurred on the most recent intrinsic call or the last FOPEN error.

The relationship between PRINTFILEINFO fields and FGETINFO, and FCHECK parameters is outlined in Figure B-9.



FGETINFO/PRINTFILEINFO

FGETINFO Parameters	PRINTFILEINTO Fields
FILENAME	FILE NAME
FOPTIONS	FOPTIONS
AOPTIONS	AOPTIONS
RECSIZE	RECORD SIZE
DEVTYPE	DEVICE TYPE,SUBTYPE
LDNUM	LDEV
HDADDR	DRT,UNIT
FILECODE	FILE CODE
RECPTR	RECPTR
EOF	EOF AT
FLIMIT	RECLIMIT

FGETINFO Parameters	PRINTFILEINFO Fields
LOGCOUNT	LOGCOUNT
PHYSCOUNT	PHYSCOUNT
BLKSIZE	BLOCK SIZE
EXTSIZE	EXTENT SIZE
NUMEXTENTS	MAX EXTENTS
USERLABELS	ULABELS
CREATORID	ID IS
LABADDR	LABEL ADDR

FCHECK/PRINTFILEINFO

FCHECK PARAMETERS	PRINTFILEINFO FIELDS
ERRORCODE	ERROR NUMBER
TLOG	RESIDUE
BLKNUM	BLOCK NUMBER
NUMRECS	NUMREC

LG200016_060

Figure B-9. Parameter/Field Relationships

Section Divider

C. Terminal Characteristics

Terminal Characteristics

Terminals and character printers, such as the HP 2613B, are supported by MPE in two modes: point-to-point and multipoint. The point-to-point mode is operated through one of three I/O controllers: the Asynchronous Terminal Controller (ATC) on the HP 3000 Series II/III, the Asynchronous Data Communications Channel (ADCC) on the HP 3000 Series 30/33/40/44, and the Advanced Terminal Processor (ATP) on the HP 3000 Series 64. (See the corresponding data sheets for devices, terminal types and other features supported on each controller.) The multipoint mode is supported by the Multipoint Terminal Software (DSN/MTS). Character printers are not supported by MPE for DSN/MTS. A full description of the DSN/MTS facility is available in the *DSN/MTS Reference Manual* (32193-90002).

This appendix deals primarily with the operation of point-to-point terminals. Most of the facilities do not apply to multipoint devices.

Terminals may be operated as session log on devices or as *file system* devices. You can control certain aspects of terminal operation with the FSETMODE, FCONTROL, and PTAPE intrinsics. Before these intrinsics can be used in a program to alter terminal characteristics, the terminal/file must be opened with the FOPEN intrinsic.

Terminals are operated in one of three modes: normal, or edited; transparent, or unedited; and binary. Normal mode is the default. In normal mode, the terminal driver provides extensive editing and control facilities to help the user make productive use of the terminal. These facilities use several keyboard-generated characters for special purposes, including one that is user-definable. These characters may not be entered into your input buffer, but are stripped from the input character stream and acted upon by the driver. Only one restriction applies to output; if the ENQ/ACK pacing handshake is enabled by means of termtype, the ENQ is considered a special character and output is suspended until the terminal replies with ACK.

Note

On the Series 30/33/40/44/64, user-embedded ENQs are not supported and may not produce the desired effect.

In transparent mode, almost all of the above facilities have been removed. Only six input special characters remain; three of these may be user-defined. These special characters are discussed later in this appendix. The ENQ is still considered a special character for output, as stated above.

In binary mode, all 256 eight-bit ASCII character patterns may be read or written. All pacing handshakes are disabled.

Table C-1 summarizes the *controlcodes* used with the **FCONTROL** intrinsic to alter terminal characteristics. These *controlcodes* are discussed in more detail in the rest of this appendix.

Table C-1. Codes for Use with FCONTROL

2	Complete input/output
3	Read hardware status word
4	Set time-out interval
10	Change terminal input speed
11	Change terminal output speed
12	Turn echo facility on
13	Turn echo facility off
14	Disable the system break function
15	Enable the system break function
16	Disable the subsystem break function
17	Enable the subsystem break function
18	Disable tape mode option *
19	Enable tape mode option *
20	Disable the terminal input timer
21	Enable the terminal input timer
22	Read the terminal input timer
23	Disable parity checking
24	Enable parity checking
25	Define line-termination characters for terminal input
26	Disable binary transfers
27	Enable binary transfers
28	Disable user block mode transfers
29	Enable user block mode transfers
34	Disable line deletion echo suppression
35	Enable line deletion echo suppression
36	Set parity * *
37	Allocate a terminal
38	Set terminal type
39	Obtain terminal type information
40	Obtain terminal output speed
41	Set unedited terminal mode
43	Abort pending NO WAIT I/O request
	* Not supported on the Series 30/33/40/44/64 computers.
	* * On the Series II/III, this enables parity generation, but not parity checking; you must issue an **FCONTROL** 23 or 24 to control parity checking. On the Series 30/33/40/44/64, this returns the current parity, but enables neither parity generation nor parity checking; use **FCONTROL** 23 or 24 to control both.

Allocating a Terminal

A terminal can be removed from speed-sensing mode, initialized according to the type and speed specified by the FCONTROL intrinsic, and set online (The terminal cannot be configured as JOB or DATA accepting.)

The format for this application of the FCONTROL intrinsic is:

```
                IV          IV        L
   FCONTROL (filenum,controlcode,param);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 37.

param — *Logical (required).* A logical word:

Bits (0:11) - Speed in characters per second.

Bits (11:5) - Terminal type (see Table C-2).

If *param* is set to zero, the speed and terminal type specified when the system was configured will be used to initialize the device. (In this case, the use of FCONTROL is not necessary; the terminal is automatically allocated when the file is opened.)

For more information about the FCONTROL intrinsic, see the *MPE V Intrinsics Reference Manual* (32033-90007).

Terminal Type Specification

MPE has limited facilities to support the features of specific terminals or devices. Originally, these facilities supported specific terminal models; on more recent machines, they have been generalized to support devices of the terminal's class. For non-HP terminals, no guarantee of successful operation is made. The facilities are designed to allow operation of the most commonly used devices.

The terminal type can be changed with the FCONTROL intrinsic. The format for this application of FCONTROL is:

```
                IV          IV        L
   FCONTROL (filenum,controlcode,param);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 38.

param — *Logical (required).* A logical word which specifies the desired terminal type (see Table C-2).

To determine the current terminal type, use the FCONTROL intrinsic with a *controlcode* of 39.

This application of FCONTROL may be used before a terminal is allocated to return the terminal type specified when the system was configured; a value of 31 is returned in *param* if no terminal type was specified at configuration time.

The format for this application of the FCONTROL Intrinsic is:

```
                IV          IV          L
   FCONTROL (filenum, controlcode,param);
```

The parameters are:

filenum *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode *Integer by value (required).* The integer 39.

param *Logical (required).* The identifier to which the terminal type is returned.

Speed and Parity Sensing

When you establish a session from a terminal, MPE uses the carriage return character that you input during the log on process to sense the line speed and parity setting of your terminal.

The ATC (Series II/III) will detect the line speed at all supported speeds. The ADCC (Series 30/33/40/44) and ATP (Series 64) are able to detect the line speed at speeds of 2400 bits per second or less; logging at higher speeds is possible only when you use the non-speed sense configuration option, subtype 4.

Only a single parity bit is available for parity sensing. The ATC, ADCC, and ATP make different assumptions based upon this bit, as shown in Table C-2.

Table C-2. Parity Sensing with the ATC, ADCC, and ATP

Party Bit on Carriage Return (15%) Is:		
	0	1
ATC	7-bit characters with odd parity are assumed; odd parity is generated on output; input checking is not done unless explicitly enabled.	7-bit characters with even parity are assumed; even parity is generated on output; input checking is not done unless explicitly enabled.
ADCC or ATP	8-bit characters are assumed; the 8th bit is passed through in both input and output.	7-bit characters with even parity are assumed. Even parity is both generated and checked.

Obtaining Terminal Output Speed

The terminal output speed can be determined with the FCONTROL intrinsic.

This application of FCONTROL may be used before a terminal is allocated to return the speed at which the device was last operated, or the speed specified when the system was configured. A value of zero is returned in *param* if the device has not been speed sensed.

The format for this application of the FCONTROL intrinsic is:

```
                IV        IV          L
   FCONTROL (filenum,controlcode,param);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 40.

param — *Logical (required).* A logical identifier to which the terminal output speed in characters per second is returned.

Changing Terminal Speed

The initial terminal speed is set either by speed sensing or by the configuration default. You can programmatically change this speed with the FCONTROL intrinsic. This capability allows a user running a mark sense card reader coupled to a terminal to operate the two devices at different speeds (for example, the card reader at 240 characters per second for input and the terminal at 10 characters per second for output).

Note

The ATC allows the input line speed to differ from the output line speed. This facility is available only on the Series II/III.

The format for this application of the FCONTROL intrinsic is:

```
               IV         IV          L
   FCONTROL (filenum,controlcode,speed);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal for which the speed is to be changed.

controlcode — *Integer by value (required).* The decimal integer 10 to change the input speed or 11 to change the output speed.

speed — *Logical (required).* A word identifier that specifies the new speed desired: 10, 14, 15, 30, 60, 120, 240, 480, or 960 characters per second. When the FCONTROL intrinsic is executed, the previous input or output speed is returned the calling process through this parameter.

As an example, consider the terminal identified by the file number stored in the word TERMFN. To change its input speed from 60 to 120 characters per second, the following call could be used. The word SPEED contains the value 120:

```
FCONTROL (TERMFN,10,SPEED);
```

After the intrinsic is executed, the word SPEED Contains the integer 60 (the previous speed).

Control of Parity Generation and Checking

All ATC controller ports are initially set with parity checking disabled. They may, however, be programmatically enabled for parity checking with the FCONTROL intrinsic. If a parity error is detected, an error code is made available through the FCHECK intrinsic.

Setting Parity

Default output parity generation is determined by the parity sensing facility. If the device is opened as a File System device (not a log on or session device), the default parity settings are used: odd for ATC, none for ADCC or ATP.

You may programmatically change both the parity type and the generation and checking facility. Note that parity generation and checking is an option only with 7-bit terminal types.

The FCONTROL intrinsic can be used to specify the parity, if any, to be used in transmitting data to a terminal. Parity is generated on the right seven bits of a character.

The format for this application of the FCONTROL intrinsic is:

```
                IV        IV          L
FCONTROL (filenum,controlcode,param);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 36.

param — *Logical (required).* A logical word, as shown in Table C-3.

Table C-3. Setting Parity for ATC or for ADCC/ATP

ATC (Series II/III)	ADCC (Series 30/33/40/44) or ATP (Series 64)
0 Output: All 8 bits are transmitted. Input: No checking; bit 8 set to 0.	Input and output: All 8 bits transmitted.
1 Output: Bit 8 set to 1. Input: No checking; bit 8 set to 0.	Input and output: All 8 bits transmitted.
2 Output: Even parity is generated if bit 8 of the output character is 0); odd parity is generated if bit 8 of the output character is 1. Input: Even parity is checked, if enabled.	Output: Even parity is generated, if enabled. Input: Even parity is checked, if enabled.
3 Output: Odd parity is generated. Input: Odd parity is generated, if enabled.	Output: Odd parity is generated, if enabled. Input: Odd parity is checked, if enabled.

Enabling and Disabling Parity Generation and Checking

This may be accomplished by using the FCONTROL intrinsic. The format for this application of FCONTROL is:

```
               IV          IV           L
FCONTROL (filenum,controlcode,anyinfo) ;
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 24 to enable parity checking, or 23 to disable parity checking.

anyinfo — *Logical (required).* Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirement of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

Setting a Time-Out Interval

You can use the FCONTROL intrinsic to apply a time-out interval on input from a terminal. If input is requested from the terminal but is not received in the specified interval, the requesting FREAD terminates at the end of the time-out interval with condition code CCL. In this case, no data is transferred to your buffer. Note that this FCONTROL affects only the next read. For block mode operation, the timer is halted when the DC2 character (CONTROL-R) is received.

The format for this application of the FCONTROL intrinsic is:

```
               IV          IV       L
FCONTROL  (filenum,controlcode,time);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 4.

time — *Logical (required).* A word identifier specifying the time-out interval in seconds. If this interval is zero, any previously established interval is cancelled, and no time-out occurs.

Read Duration Timer

The terminal input timer records the time required to satisfy an input request on the terminal, from the time the input is requested until it is completed. This applies only to unbuffered, serial terminal input requests.

You can programmatically enable or disable the terminal input timer with the FCONTROL intrinsic.

The format for this application of the FCONTROL intrinsic is:

```
            IV          IV          L
FCONTROL  (filenum,controlcode,anyinfo);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 21 to enable the timer, or 20 to disable the timer.

anyinfo — *Logical (required).* Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

Reading the Terminal Input Timer

You can read the result from the terminal input timer with the FCONTROL intrinsic. The result will be valid only if the terminal input was preceded by a call to enable the terminal input time. If valid, the result is the time, in hundredths of seconds, required for the last direct, unbuffered serial input on the terminal.

The format for this application of the FCONTROL intrinsic is:

```
                 IV          IV          L
FCONTROL  (filenum,controlcode,inputtime);
```

The parameters are:

filenum *Integer by value (required).* A word identifier supplying the number of the terminal.

controlcode *Integer by value (required).* The integer 22.

inputtime *Logical (required).* A word to which the input time is returned (in hundredths of seconds).

Figure C-1 contains a program that generates an ASCII character, instructs the user to enter this character on the terminal, then measures and displays the reaction time of the user.

At line 26, the statement:

```
FCONTROL (IN,21,DUMMY);
```

enables the terminal input timer so that the reaction time of the user can be measured. The parameter IN supplies the file number of the terminal and was obtained through the FOPEN intrinsic call (see statement 19 in the program).

At line 28, the statement:

```
FCONTROL (IN,4,TIME OUT);
```

is used to set a time-out interval of ten seconds (see statement 5 in the program). If there is no response to the FREAD intrinsic call (statement 33) within ten seconds, a CCL condition code is returned and the program displays the message:

```
YOU'RE TOO SLOW!
```

At line 45, the statement:

```
FCONTROL (IN,22,TIME);
```

reads the reaction time from the terminal input timer. This result is returned to the word TIME.

At line 47, the statement:

```
ASCII (TIME*10,10,CRESP (15);
```

multiplies the value of TIME by ten and converts this result to an ASCII string so that the user's reaction time, in milliseconds, can be displayed. The resulting ASCII string is stored in the byte array CRESP,starting at the 16th position (CRESP (15)).

At line 48, the statement:

```
FWRITE (OUT,RESPONSE,17,0);
```

displays the reaction time. (Arrays CRESP and RESPONSE have been equivalenced; see statements 12 and 13.)

```
PAGE 0001 HEWLETT-PACKARD 32100A.05.01
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INNAME(0:5) :="INPUT ";
00004000 00004 1 BYTE ARRAY OUTNAME(0:6):="OUTPUT ";
00005000 00000 0 INTEGER IN,OUT,LGTH,DUMMY,TIME,TIMEOUT:=10;
00006000 00000 1 ARRAY BUFR(0:3):="TYPE X",0;
00007000 00004 1 BYTE ARRAY CBYF(*)=BUFR;
00008000 00004 1 ARRAY INSTRUCTIONS(0:34):="REACTION TIMER: ",%6412,
00009000 00011 1 "TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN. ";
00010000 00043 1 ARRAY MSG(0:24):=:TRY AGAIN? (Y/N)","WRONG CHARACTER.",
00011000 00020 1 %6412,"YOU'RE TOO SLOW!";
00012000 00031 1 ARRAY RESPONSE(0:16):="REACTION TIME:MILLISECONDS";
00013000 00021 1 BYTE ARRAY CRESP(*)=RESPONSE;
00014000 00021 1
00015000 00021 1 INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,ASCII,TIMER,QUIT;
00016000 00021 1
00017000 00021 1
```

END OF DECLARATIONS

```
00018000 00021 1
00019000 00021 1 IN:=FOPEN(INNAME,%45; <<&STDIN>>
00029000 00007 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00021000 00012 1 OUT:=FOPEN(OUTNAME,%414,%1); <<$STDLIST>>
00022000 00022 1 IF < THEN QUIT(2); <<CHECK FOR ERROR>>
00023000 00025 1 FWRITE(OUT,INSTRUCTIONS,35,0); <<user ul DIRECTIONS>>
00024000 00032 1 IF < THEN QUIT(3; <<CHECK FOR ERROR>>
00025000 00035 1 LOOP:
00026000 00035 1 FCONTROL(IN,21,DUMMY; <<ENABLE TIMER READ>>
00027000 00041 1 IF < THEN QUIT(4); <<CHECK FOR ERROR>>
00028000 00044 1 FCONTROL(IN,4,TIMEOUT); <<ENABLE TIMEOUT>>
00029000 00050 1 IF < THEN QUIT(5): <<CHECK FOR ERROR>>
00030000 00053 1 CBUF(5):=INTEGER(TIMER),(11:5)+%73;
         <<GENERATE A CHARACTER>>
00031000 00062 1 FWRITE(OUT,BUFR,3,%320); <<REQUEST USER INPUT>>
00032000 00067 1 IF < THEN QUIT(6); <<CHECK FOR ERROR>>
00033000 00072 1 LGTH:=FREAD(IN,BUFR(3),-1; <<READ CHARACTER>>

00034000 00101 1 IF < THEN <<TIMEOUT OCCURRED>>
00035000 00102 1 BEGIN
00036000 00102 2 FWRITE<OUT,MSG(16),9,8>; <<TOO SLOW MESSAGE>>
00037000 00110 2 IF < THEN QUIT(7) ELSE GO NEXT; <<CHECK FOR ERROR>>
00038000 00120 2 END:
00039000 00120 1 IF CBUF(5)<>CBUF(6) THEN <<INCORRECT CHARACTER>>
00040000 00126 1 BEGIN
```

```
00041000 00126 2 FWRITE<OUT,MSG(8),8,0>; <<WRONG CHARACTER MESSAGE>>
00042000 00134 2 IF < THEN QUIT(8) ELSE TO NEXT; <<CHECK FOR ERROR>>
00043000 00141 2 END:
00044000 00141 1 MOVE RESPONSE (7) :=" "; <<RESET RESPONSE TIME>>
00045000 00153 1 FCONTROL(IN,22,TIME); <<READ INPUT TIME>>
00046000 00157 1 IF <> THEN QUIT(9) <<CHECK FOR ERROR>>
00047000 00162 1 ASCII<TIME*10,10,CRESP(15)>; <<CONVERT TIME>>
00048000 00171 1 FWRITE(OUT,RESPONSE,17,0); <<REACTION TIME>>
00049000 00177 1 IF < THEN QUIT(10); <<CHECK FOR ERROR>>
00050000 00202 1 NEXT:
00051000 00202 1 FWRITE(OUT,MSG,8,%320); <<CONTINUE TEST>>
00052000 00207 1 IF < THEN QUIT(11) <<CHECK FOR ERROR>>

00053000 00212 1 FREAD(IN,BUFR(3) ,-1); <<GET Y/N ANSWER>>

00054000 00220 1 IF < THEN QUIT(12); <<CHECK FOR ERROR>>
00055000 00224 1 IF CBUF(6)="Y" THEN GO LOOP; <<Y-CONTINUE TEST>>
00056000 00232 1 END
  PRIMARY DB STORAGE=%016; SECONDARY DB STORAGE=%00130
  NO. ERRORS=000; NO. WARNING+000
  PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:10
```

Figure C-1. Using FCONTROL to Enable/Read title the Terminal Input Timer (1 of 2)

A sample run of the program of Figure C-1 is shown below. User input is underlined in this example:

```
:RUN TIME

REACTION TIMER
TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN
TYPE M
YOU'RE TOO SLOW!
TRY AGAIN? (Y/N) Y
TYPE>>
REACTION TIME: 9670 MILLISECONDS
TRY AGAIN? (Y/N) Y
TYPE UU
REACTION TIME: 4090 MILLISECONDS
TRY AGAIN? (Y/N) Y
TYPE BB
REACTION TIME: 1790 MILLISECONDS
TRY AGAIN? (Y/N) Y
TYPE IO
WRONG CHARACTER
TRY AGAIN? (Y/N) N

END OF PROGRAM
:
```

End of Record Characters

Normally, when using a terminal, you indicate the end of a line by entering a carriage return (with the RETURN key on most terminals). With the FCONTROL intrinsic, however, you can specify that an additional character, such as an equal sign, a period, or an exclamation point, be recognized as a line terminator. On subsequent read operations to the *filenum* specified in your FCONTROL Call, the input operation is terminated by the specified character: receipt of this character causes MPE to terminate an FREAD and return to your program. The character is returned to your buffer. No carriage return or line feed is generated.

The format for this application of the FCONTROL intrinsic is:

```
                IV        IV         L
FCONTROL (filenum,controlcode,character);
```

The parameters are:

filenum *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode *Integer by value (required).* The integer 25.

character *Logical (required).* A word identifier supplying (in the right byte) the character to be used as a line terminator. The left byte of this word can contain any information &—; it is ignored by the intrinsic. If the character null (%0) is specified in the *character* parameter, the terminal reverts to its normal line-control operation.

The following characters are not recognized as line-terminating characters during normal reads:

ASCII Character		**Octal Code**
Backspace	(CONTROL-H)	%10
Line Feed	(CONTROL-J)	%12
Carriage Return	(CONTROL-M)	%15
X-ON	(CONTROL-Q)	%21
DC2	(CONTROL-R)	%22
X-OFF	(CONTROL-S)	%23
Line Delete	(CONTROL-X)	%30
CONTROL-Y	(CONTROL-Y)	%31
Escape	(CONTROL-[)	%33
Del		%177

In addition, when you are working at the Console, CONTROL-A will not be recognized as a line terminator.

As an example, to specify a period as an additional line terminator for a terminal, the following intrinsic call could be used:

```
FCONTROL (TERMFN,25,CHAR);
```

The word CHAR contains the octal value %56 (indicating a period) in the right byte. The left byte can contain any value.

Break Functions

With the FCONTROL intrinsic, you can enable and disable the system and subsystem break functions.

Enabling and Disabling System Break Function

You can programmatically enable or disable a terminal's ability to generate a system break request with the FCONTROL intrinsic. (The default is for this ability to be enabled.) System break requests are initiated by pressing the BREAK key or by calling the CAUSEBREAK intrinsic.

The format for this application of the FCONTROL intrinsic is:

```
                IV       IV          L
FCONTROL (filenum,controlcode,anyinfo);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 15 to enable the break function, or 14 to disable the break function.

anyinfo — *Logical (required).* Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of the intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

As an example, to enable the break function, the following intrinsic call could be used:

```
FCONTROL (TERMFN,15,DUMMY);
```

Note

Using FCONTROL to disable break does not affect operation of the CAUSEBREAK intrinsic.

Enabling and Disabling Subsystem Break Function

All terminals are initially set to disable (not accept) subsystem break requests, generated by entering CONTROL-Y during a session. You can, however, programmatically enable and again disable a terminal's ability to generate subsystem break requests with the FCONTROL intrinsic.

The format for this application of the FCONTROL intrinsic is:

```
                IV       IV          L
FCONTROL (filenum,controlcode,anyinfo);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 17 to enable the sub-system break function, or 16 to disable the subsystem break function.

anyinfo — *Logical (required).* Any variable or word identifier. This parameter is needed by `FCONTROL` to satisfy the internal requirements of the intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

As an example, to enable the subsystem break function, the following intrinsic call could be used:

```
   FCONTROL (TERMFN,17,DUMMY);
```

Note

For more information about the CONTROL-Y trap, consult the `XCONTRAP` intrinsic in the *MPE V Intrinsics Reference Manual* (32033-90007).

Operating in Normal Mode

During input (using `FREAD`, `READ`, or `READX`), a number of characters and character sequences have special meanings to MPE. These characters are listed in Table C-4.

Note

In Table C-4, the superscript c denotes a control character. Thus “X^c”, means “CONTROL-X.” These descriptions may be used interchangeably.

Table C-4. Special Characters

Character	Meaning
A^c	When you are operating from the System Console, A^c initiates a Console command.
H^c (backspace)	Deletes the previous character. (To delete *n* characters, enter H^c *n* times.)
J^c (LF,linefeed)	For any terminal with a linefeed entry, you may strike this key and a carriage return will be echoed. The linefeed character is not placed in the input buffer. This mechanism is primarily intended for devices which do not have an automatic line wraparound feature. For reads of length greater than the device's line width, LFs may be included so that the input will be displayed on several lines on the device. thus avoiding overstrike of characters in the last column position of the device.
M^c (CR, carriage return)	Normal end-of-record character.
Q^c (DC1, X-ON)	Places terminal in tape mode, allowing input from paper tape. This facility is supported only on the Series II/III. When enabled, the tape-mode option inhibits the implicit linefeed normally issued by MPE each time a carriage return is entered. The tape-mode option also inhibits responses to H^c and X^c entries. Thus, when X^c is received and tape mode is in effect, no exclamation points (!!!) are sent to the terminal. Tape mode is terminated by Y^c. If used after S^c, Q^c also resumes write operation during output (cancels S^c).
R^c (DC2)	Indicates the beginning of a block mode read and starts a special block mode timer. If the read does not complete successfully within the timer period, the read is returned with an FSERR 27. Normal block mode transfers proceed as follows: The computer sends DC1 to the terminal to initiate a read. If the user has pressed ENTER for a block mode read, the terminal then sends DC2 (R^c) to the computer to indicate a block mode read; the computer sends another DC1 to the terminal to initiate the transfer; the terminal then sends the data to the computer. NOTE: R^c has special significance only for termtypes which support block mode.
SC^c (DC3, X-OFF)	Suspends the write operation during output. Output may be resumed with Q^c.

Table C-4. Special Characters (continued)

Character	Meaning
X^c	Deletes (Ignores) all characters read on this line and restarts the read. The system responds with a triple exclamation point (!!!) followed by a carriage return and linefeed.
Y^c	If the terminal is not in tape mode, Y^c requests subsystem break. If the terminal is in tape mode, Y^c returns it to the keyboard mode.
BREAK	Requests a system break.
(ESC):	Places the terminal in the echo-on mode so that characters input are echoed on the terminal by MPE. NOTE: (ESC) indicates the ESCAPE key on your terminal keyboard.
(ESC);	Place the terminal in echo-off mode so that characters input are not echoed on the terminal by MPE.

The defined control characters A^c, H^c, Q^c, S^c, X^c, Y^c, CR, and LF are recognized even when following an (ESC) key entry. However, entry of (ESC) followed by any other character (other than one of these control characters, or a semicolon) is read as a two character string in your input stream.

Enabling and Disabling User Block Transfers

User mode block transfers (from block mode terminals such as the HP 2644/2645) can be enabled or disabled with the FCONTROL intrinsic. User mode block transfers are disabled in normal MPE operation. The DC2 (CONTROL-R), transmitted by the terminal when you press ENTER, is passed to your program for action At this point you may write escape sequences to the terminal (i.e., to position the cursor) before reading the data from the terminal.

The format for this application of the FCONTROL intrinsic is:

```
                IV          IV          L
FCONTROL (filenum,controlcode,anyinfo);
```

The parameters are:

filenum *Integer by value (required).* A word identifier supplying the file number of the terminal

controlcode *Integer by value (required).* The integer 28 to disable user mode block transfers, or 29 to enable user mode block transfers.

anyinfo *Logical (required).* Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

Note

Data overruns may occur during block mode transfers. Your applications programs must check for successful completion of each `FREAD` operation and retry as required. Since a data overrun on the last character read will cause the port to hang on the ADCC (Series 30/33/40/44). The normal block read timer will not work for *own* handshaking.

Changing Input Echo Facility

You can programmatically determine whether MPE transmits (echoes) input from the terminal keyboard back to the terminal display by calling the `FCONTROL` intrinsic to turn the echo facility on or off.

When the echo facility is on, input read from the terminal is echoed to the terminal by the terminal controller hardware. If the terminal is operating in full-duplex mode, the echoed information appears as normal printed lines. If the terminal is in half-duplex mode on a full-duplex line, however, the echoed printing may be illegible. As you enter input on such terminals, it is simultaneously printed by the terminal itself and subsequently overwritten by the echoed information. When you log on, all terminals are assumed to be in the full-duplex mode.

The format for this application of the `FCONTROL` intrinsic is:

```
                IV          IV        L
FCONTROL (filenum,controlcode,last);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 12 to turn the echo facility on, or 13 to turn it off.

last — *Logical (required).* A word identifier to which the previous echo status is returned, where:

0 = Echo on.
1 = Echo off.

As an example, to turn the echo facility off, the following intrinsic call could be used:

```
FCONTROL(TERMFN,13,LAST);
```

After the intrinsic is executed, the word `LAST` contains the value 0 or 1 to reflect the previous echo facility status.

Note

In addition to the `FCONTROL` intrinsic, the echo facility can be switched on and off by entering the following two-character sequences from your terminal:

(ESC) (:) = To turn the echo facility on.
(ESC) (;) = To turn the echo facility off.

Enabling and Disabling Tape-Mode Option

For Series II/III only you can programmatically enable or disable the tape-mode option for a terminal with the FCONTROL intrinsic. When enabled, the tape-mode option inhibits the implicit line feed normally issued by MPE each time a carriage return is entered. The tape mode option also inhibits responses to CONTROL-H and CONTROL-X entries. Thus, when CONTROL-X is received and tape mode is in effect, no exclamation points (!!!) are sent to the terminal. To inhibit carriage return and/or linefeed for FREAD, use the FSETMODE intrinsic (see the *MPE V Intrinsics Reference Manual* (32033-90007).

The format for this application of the FCONTROL intrinsic is:

```
                IV        IV        L
FCONTROL (filenum,controlcode,anyinfo);
```

The parameters are:

finenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 19 to enable tape mode, or 18 to disable tape mode.

anyinfo — *Logical (required).* Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

As an example, the following intrinsic call could be used to enable tape mode:

```
FCONTROL(TERMFN,19,DUMMY);
```

Enabling and Disabling Line Deletion Echo Suppression

In normal MPE operation, CONTROL-X is interpreted as a line deletion character, and the character string "!!!" is printed on the terminal when it is used. You can suppress the line deletion echo, so that the character string is not displayed on the terminal, with the FCONTROL intrinsic.

Note

This application of FCONTROL disables only the "!!!" string; it does not disable the line deletion operation.

The format for this application of the FCONTROL intrinsic is:

```
                IV          IV          L
FCONTROL (filenum,controlcode,anyinfo);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode	*Integer by value (required).* The integer 34 to disable the line deletion echo, or 35 to enable the line deletion echo.
anyinfo	*Logical (required).* Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic, however, it serves no other purpose and is not modified by the intrinsic.

Reading Paper Tapes Without X-OFF Control

The X-OFF control character, written by pressing the X-OFF key on a teletype terminal, is used to delimit data input on paper tape. When a teletype tape reader encounters this character while reading a tape, reading halts until the program requests more input data.

You can programmatically read data from paper tapes not containing the X-OFF control character, or from tapes input through terminals not recognizing this character, with the PTAPE Intrinsic. In the latter case, the X-OFF characters are stripped from the tape. Tape input terminates when CONTROL-Y is encountered, returning control to the terminal. Prior to calling the PTAPE intrinsic, you must be sure to position the endfile pointer in the disk file to the proper position. If you are reading more than one tape, you should set the disk file's end-of-file pointer to zero, if necessary, before issuing the first PTAPE intrinsic call.

A PTAPE intrinsic call such as:

```
PTAPE(TEMFN,DISCFL);
```

could be used to read a paper tape not containing the X-OFF control character, or to read a paper tape input through a terminal that does not recognize this character. The data would be stored in the disk file whose file number is specified by DISCFL.

To inhibit carriage return and linefeed after FREAD, use the FSETMODE intrinsic (see the *MPE V Intrinsics Reference Manual* (32033-90007).

Operating in Transparent (Unedited) Mode

Your terminal can be set in unedited mode with the FCONTROL intrinsic. In unedited mode, all characters, except those specified below, are passed to your input buffer.

The end-of-record character terminates input from the terminal in unedited mode, as a carriage return does in normal mode.

If enabled, the Attention character terminates input and causes a Subsystem Break in unedited mode, as a CONTROL-Y does in normal mode.

For block-mode terminal types, input of a DC2 (CONTROL-R) as the first character, r embedding the character pair DC2CR anywhere in the data stream, causes those characters to be stripped out and a DC1 (CONTROL-Q) to be written.

For the logical System Console, CONTROL-A (Start of Header) signals the beginning of a Console command.

The X-ON/X-OFF (DC1/DC3) handshake characters are assumed to be protocol characters and are stripped from the input stream.

Note

If a terminal is accessed with `FOPEN` multiple times, and `FCONTROL` is used to set unedited mode for any of the terminal files, unedited mode will be in effect on all of the terminal files.

No automatic linefeed is output to the terminal when input terminates in unedited mode.

In unedited mode, only the ENQ character has special consequences on output. For terminals doing the ENQ/ACK handshake, output is suspended following an ENQ to wait for an ACK from the terminal; generally, the terminal strips the ENQ from the data stream.

The unedited mode is reset to normal when an `FCLOSE` intrinsic call is issued against the terminal, or when the chars parameter of `FCONTROL` equals zero. (See below.)

The unedited mode is disabled while the terminal is in Break or Console mode.

Use `FCONTROL` to set unedited terminal mode:

```
                IV           IV          L
FCONTROL  (filenum,controlcode,chars);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 41.

chars — *Logical (required).* A logical word, as follows:

Bits (0:8) - Attention character.

Bits (8:8) - End-of-record character.

If *chars* = 0, the unedited mode is reset to normal.

Operating in Binary Mode

Binary transfers can be enabled or disabled with the `FCONTROL` Intrinsic. (By default, binary transfers are disabled in normal MPE operation.) Binary reads are terminated only by the Read Byte Count or by the Read Time Out.

The format for this application of the `FCONTROL` intrinsic is:

```
               IV        IV      L
FCONTROL  (filenum,controlcode,anyinfo);
```

The parameters are:

filenum — *Integer by value (required).* A word identifier supplying the file number of the terminal.

controlcode — *Integer by value (required).* The integer 26 to disable binary transfers, or 27 to enable binary transfers.

anyinfo — *Logical (required).* Any variable or word identifier. This parameter is needed by `FCONTROL` to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

Section Divider

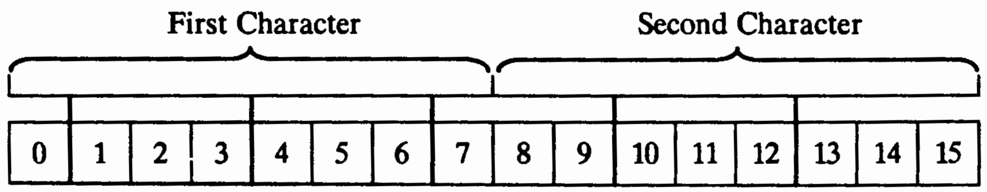
D. ASCII Character Set

ASCII Character Set

Character	Octal Equivalent	Octal Equivalent
A	040400	000101
B	041000	000102
C	041400	000103
D	042000	000104
E	042400	000105
F	043000	000106
G	043400	000107
H	044000	000110
I	044400	000111
J	045000	000112
K	045400	000113
L	046000	000114
M	046400	000115
N	047000	000116
0	047400	000117
P	050000	000120
Q	050400	000121
R	051000	000122
S	051400	000123
T	052000	000124
U	052400	000125
V	053000	000126
W	053400	000127
X	054000	000130
Y	054400	000131
Z	055000	000132
a	060400	000141
b	061000	000142
c	061400	000143
d	062000	000144
e	062400	000145
f	063000	000146
g	063400	000147
h	064000	000150
i	064400	000151
j	065000	000152
k	065400	000153
l	066000	000154

Character	Octal Equivalent	Octal Equivalent
m	066400	000155
n	067000	000156
o	067400	000157
p	070000	000160
q	070400	000161
r	071000	000162
s	071400	000163
t	072000	000164
u	072400	000165
v	073000	000166
w	073400	000167
x	074000	000170
y	074400	000171
z	075000	000172
0	030000	000060
1	030400	000061
2	031000	000062
3	031400	000063
4	032000	000064
5	032400	000065
6	033000	000066
7	033400	000067
8	034000	000070
9	034400	000071
NUL	000000	000000
SOH	000400	000001
STX	001000	000002
ETX	001400	000003
EOT	002000	000004
ENQ	002400	000005
ACK	003000	000006
BEL	003400	000007
BS	004000	000010
HT	004400	000011
LF	005000	000012
VT	005400	000013
FF	006000	000014
CR	006400	000015
SO	007000	000016
SI	007400	000017
DLE	010000	000020
DC1	010400	000021
DC2	011000	000022
DC3	011400	000023
DC4	012000	000024
NAK	012400	000025

Character	Octal Equivalent	Octal Equivalent
SYN	013000	000026
ETB	013400	000027
CAN	014000	000030
EM	014400	000031
SUB	015000	000032
ESC	015400	000033
FS	016000	000034
GS	016400	000035
RS	017000	000036
US	017400	000037
SPACE	020000	000040
!	020400	000041
”	021000	000042
#	021400	000043
$	022000	000044
%	022400	000045
&	023000	000046
,	023400	000047
(	024000	000050
)	024400	000051
*	025000	0000
+	025400	000053
‘	026000	000054
-	026400	000055
.	027000	000056
/	027400	000057
:	035000	000072
;	035400	000073
<	036000	000074
=	036400	000075
>	037000	000076
?	037400	000077
@	040000	000100
[	055400	000133
\	056000	000134
]	056400	000135
ˆ	057000	000136
_	057400	000137
‘	060000	000140
{	075400	000173
|	076000	000174
}	076400	000175
˜	077000	000176
DEL	077400	000177



Section Divider

E. Disk File Labels

E

Disk File Labels

Whenever a disk file is created, MPE automatically supplies a file label in the first sector of the first extent occupied by that file. Such labels always appear in the format described below. (User-supplied labels, if present, are located in the sectors immediately following the MPE file label.) The contents of a label may be listed by using the `LISTF −1` command described in the *MPE V Commands Reference Manual* (32033-90006).

Words		Contents
0-3		Local file name
4-7		Group name
8-11		Account name
12-15		User name of file creator
16-19		File lockword
20-21		File security matrix
22	(Bits 0:15)	Not used
	(Bit 15:1)	File secure bit:
		If 1, file secured
		If 0, file released
23		File creation date
24		Last access date
25		Last modification date
26		File code
27		File control block vector
28	(Bit 0:1)	Store Bit (If on, STORE or RESTORE in progress)
	(Bit 1:1)	Restore Bit (If on, RESTORE in progress)
	(Bit 2:1)	Load Bit (If on, program file is loaded)
	(Bit 3:1)	Exclusive Bit (If on, file is opened with exclusive access)
	(Bits 4:4)	Device sub-type
	(Bits 8:6)	Device type
	(Bit 14:1)	File is open for write
	(Bit 15:1)	File is open for read
29	(Bits 0:8)	Number of user labels written

Words		Contents
	(Bits 8:8)	Number of user labels available
30-31		File limit in blocks
32-33		Private volume information (while file is open)
34		File label check sum (used for error detection)
35		Cold-load identity
36		*FOPTION*s specifications
37		Logical record size (in negative bytes)
38		Block size (in words)
39	(Bits 0:8)	Sector offset to data
	(Bits 8:3)	Not used
	(Bits 11:5)	Number of extents-1
40		Last extent size in sectors
41		Extent size in sectors
42-43		Number of logical records in file
44-45		First extent descriptor
46-107		Remaining extent descriptors (32 maximum)
108-109		Restore time
110		Restore date
112-113		Start of file block number
114-115		Block number of End-of-file
116-117		Number of open and close records
120-123		Not used
124-127		Device class name

Note

An extent descriptor (words 44 through 107 above) is a double-word. The first byte contains the volume table index of the volume in which the extent resides; the remaining three bytes of the double-word extent descriptor contain the first sector number of the extent.

Section Divider

F. End-of-File Indication

F

End-of-File Indication

The end-of-file indication will be returned by the card reader and tape drivers under conditions specified by the initiators of read requests. The type of requests are as follows:

Type	Class of End-Of-File
A	All records that begin with a colon (:).
B	All records that contain, starting in the first byte, EOD, EOJ, JOB and DATA. (See Note.)
E	Hardware-sensed end-of-file.

Note

If the word count is less than three or the byte count is less than six, then Type B reads are converted to Type A reads.

In utilizing the card/tape devices as files via the file system, the following types are assigned:

File Specified	Type
$STDIN	Type A.
$STDINX	Type B.
Dev=CARD/TAPE	Type B, if device job/data accepting. Type E, if device not job/data accepting.

Any subsequent requests initiated by the driver following sensing of an end-of-file condition will be rejected with an end-of-file indication.

When reading from an unlabeled tape file, the request encountering a tape mark will respond with an end-of-file indication but succeeding requests will be allowed to continue to read data past the tape mark. Under these conditions, it is the responsibility of the caller to protect against the reading of unrelated data behind an end-of-file and to prevent reading off the end of the reel.

Section Divider

G. Magnetic Tape Labels

Magnetic Tape Labels

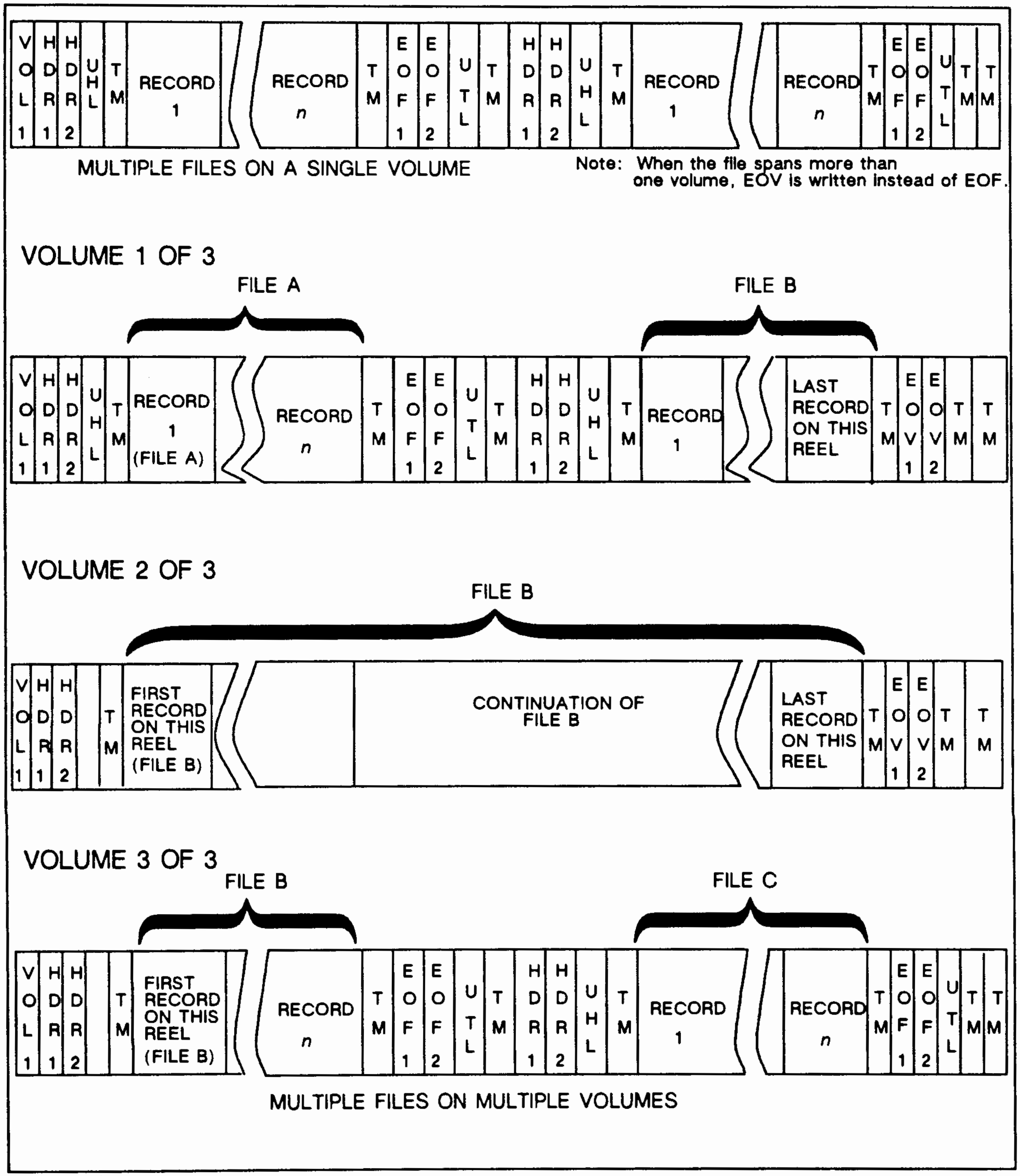
Labels conforming to ANSI-standard can be read and written on magnetic tape files by MPE. IBM-standard labels can be read, but cannot be written by MPE.

The tape labels written by MPE consist of:

Volume Header	At the beginning of each reel of tape.
File Header 1	At the beginning of each file on the reel.
File Header 2	Following File Header 1.
End-of-File 1	At the end of each file on the reel.
End-of-File 2	Following End-of-File 1.
End-of-Volume 1	At the end of a reel if the tape spans more than one volume.
End-of-Volume 2	Following End-of-Volume 1.

The file labels (file headers, end-of-file, and end-of-volume labels) are specified using the `FILE` command or the `FOPEN` intrinsic. Each label is 80 bytes long and is formatted as shown in Figure G-1 and Table G-1.

User-supplied labels, if any, are located on the tape as shown in Figure G-1. User-supplied labels can only be written on tape labeled with MPE tape labels, and the user labels must be exactly 80 bytes, to conform to the ANSI standard.



LG200016_061

Figure G-1. MPE Tape Labels (Conforming to ANSI-Standard)

Table G-1.
Format of Tape Labels Written by MPE. (ANSI Standard)

VOLUME HEADER LABEL (80 BYTES)		
POSITION	CONTENTS	COMMENTS
Bytes 1-4	VOL n	Indicates volume label (n specifies the volume number). Appears on each label.
Bytes 5-10	volume id	Six-character identifier as supplied by `FILE` command, `FOPEN` intrinsic, or Console Operator.
Bytes 11-37	Blanks	Reserved for future use.
Bytes 38-51	Blanks	Not used by MPE. (Reserved for owner identification in ANSI-standard labels.)
Bytes 52-79	Blanks	Reserved for future use.
Byte 80	1	Indicates that label conforms to ANSI-standard.

Table G-2.
Format of Tape Labels Written by MPE. (ANSI Standard)

FILE HEADER LABEL (80 BYTES)		
POSITION	CONTENTS	COMMENTS
Bytes 1-4	HDRI	Indicates file header 1 label. Appears before each file on the reel.
Bytes 5-21	filename.groupname	Used for file identifier in ANSI-standard labels.
Bytes 22-17	volume set id	Six-character identifier of the first volume in a set, as supplied by **FILE** command, **FOPEN** intrinsic, or Console Operator.
Bytes 28-31	reel number	A four-digit entry from 0001 to 9999, indicating the relative position of a reel in a volume set.
Bytes 32-35	file sequence number	A four-digit entry from 0001 to 9999, indicating the relative position of a file within a volume set.
Bytes 36-39	Generation number	Always 0001.
Bytes 40-41	Version number	Always 00.
Bytes 42-47	file creation date	Indicates date on which file is written to magnetic tape.
Bytes 48-53	file expiration date	Indicates date file can be overwritten.
Bytes 54	%230 or blank	If %230, indicates the file has a lockword.
Bytes 55-60	Block count	Written as "000000"; otherwise ignored.
Bytes 61-80	System ID	Written as "HP MPE 3000".

Table G-2.
Format of Tape Labels Written by MPE. (ANSI Standard)
(continued)

FILE HEADER LABEL (80 BYTES)		
POSITION	CONTENTS	COMMENTS
Bytes 1-3	label identifier	Indicates file header 2 label.
Byte 4	label number	normally 2
Byte 5	record format	"F" indicates a fixed record format "V" indicates a variable record format "U" indicates an undefined record format
Bytes 6-10	block length	Indicates the block length (in character format)
Bytes 11-15	record length	Indicates the record length (adhering to MPE rules) in characters.
Bytes 16-23	lockword	Indicates the MPE lockword.
Bytes 24-36	Blank	Not used by MPE.
Byte 37	record type	"A" indicates an ASCII record. "B" indicates a Binary record.
Byte 38	carriage control	"C" indicates carriage control. Blank indicates no carriage control.
Bytes 39-80	Blank	Not used by MPE.

Computer Museum

Section Divider

Index

Index

Index (Continued)

Index (Continued)

Index (Continued)

Manual Part Number 30000-90236
Printed in U.S.A. 06/87
E1089

HEWLETT
PACKARD