HP 3000 Computer Systems

HEWLETT PACKARD

MPE File System
reference manual

HP 3000 Computer Systems

MPE File System

Reference Manual

HEWLETT PACKARD

19447 PRUNERIDGE AVENUE, CUPERTINO, CA 95014

ii

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars are removed but the dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition . . . . Feb 1982

PRINTING HISTORY

First Edition .... Feb 1982

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

TABLES

INTRODUCTION

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

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

Logically related records are grouped into sets known to the file system as *files*, which may be kept in any storage medium or sent to any input-output peripheral. Since all input-output operations are done through the mechanism 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 disc, 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 is called *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 disc 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 MPE :FILE commands to specify the device you want.

Figure 1-1 shows the relationships among your program, the MPE File System, the MPE I/O System, and the actual hardware of the system. Notice that the MPE 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.
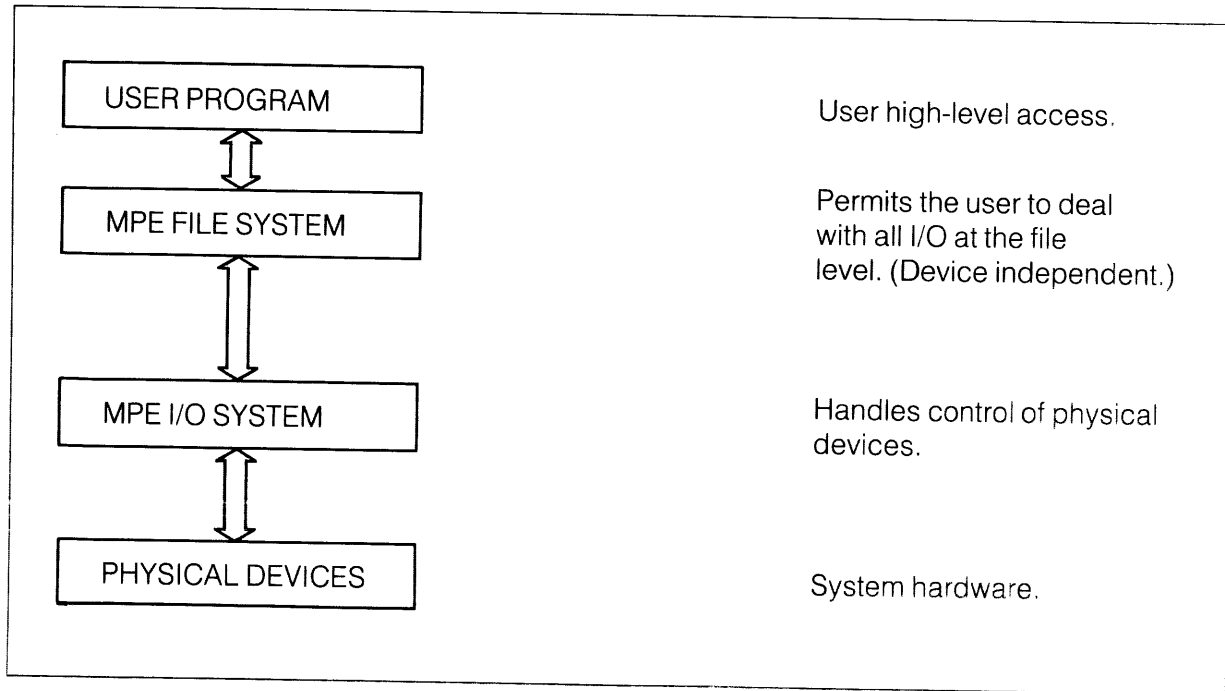
Figure 1-1.   File System Interface.

TOPICS IN THIS MANUAL

In its simplest form, information is contained in data fields which are organized into *logical records*. Section 2, Record Structure, discusses the formats you may specify for your records and describes the best approaches to efficient blocking.

Sets of logically related records are known to the file system as files. In Section 3, File Structure, we discuss file size, the arrangement of files in extents, file identification, and the specification of file characteristics.

Your file may be classified as a new, temporary or permanent file. Section 4, Domains, discusses these classifications.

How do you define and use the files you create? Section 5, File Operation, covers the usage and operation of your files.

One of the file system's principal concerns is the transfer of information to and from your files. Section 6, Data Transfer, discusses the selection of records and the transfer of information, including the use of buffers and considerations for shared files.

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 or group, or to that particular file. These provisions are discussed in Section 7, File Security.

Interprocess communication (IPC) is a facility of the file system which permits multiple user processes to communicate with one another in an easy and efficient manner. Section 8, Interprocess Communication, describes this facility.

The most common medium for offline file storage is magnetic tape. Section 9, Magnetic Tape Considerations, discusses the matters you should bear in mind as you work with your magnetic tape files.

The appendices of this manual contain useful reference information. Appendix A, File System Reference, summarizes much of this manual's information.

As you work with disc files, occasionally you will want to check their status. You can investigate physical characteristics, current file information, and error information; Appendix B, Status Information, describes how to do this.

Much of your work may involve the use of terminals. Appendix C, Terminal Characteristics, contains much useful information about these devices.

The last few appendices contain brief summaries of information. Appendix D contains the ASCII character set, Appendix E describes the contents of disc file labels, Appendix F discusses the end-of-file indication, and Appendix G describes magnetic tape labels.

RECORD STRUCTURE AND BLOCKING

When logically related data elements are grouped together, they form a **logical record**. This is the fundamental unit of information that is handled by the MPE File System: it is the smallest data grouping that can be identified 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 section we will discuss these questions.

DATA REPRESENTATION

ASCII vs. Binary

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

NOTE

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

With many devices, there is no restriction on the data actually transferred to or from the file: you can write ASCII data to a binary file or binary data to an ASCII file. You can specify the type of code 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 zeroes for binary files; this padding is the only real 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.

RECORD FORMATS

A file can contain records written in 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 the MPE :BUILD or :FILE command. FOPEN, :BUILD, and :FILE are discussed in the section on File Structure. For more detailed coverage of the FOPEN intrinsic, consult the MPE Intrinsics Reference Manual, part number 30000-90010; for more detailed coverage of the :BUILD and :FILE commands, consult the MPE Commands Reference Manual, part number 30000-90009.

Files residing on disc 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.
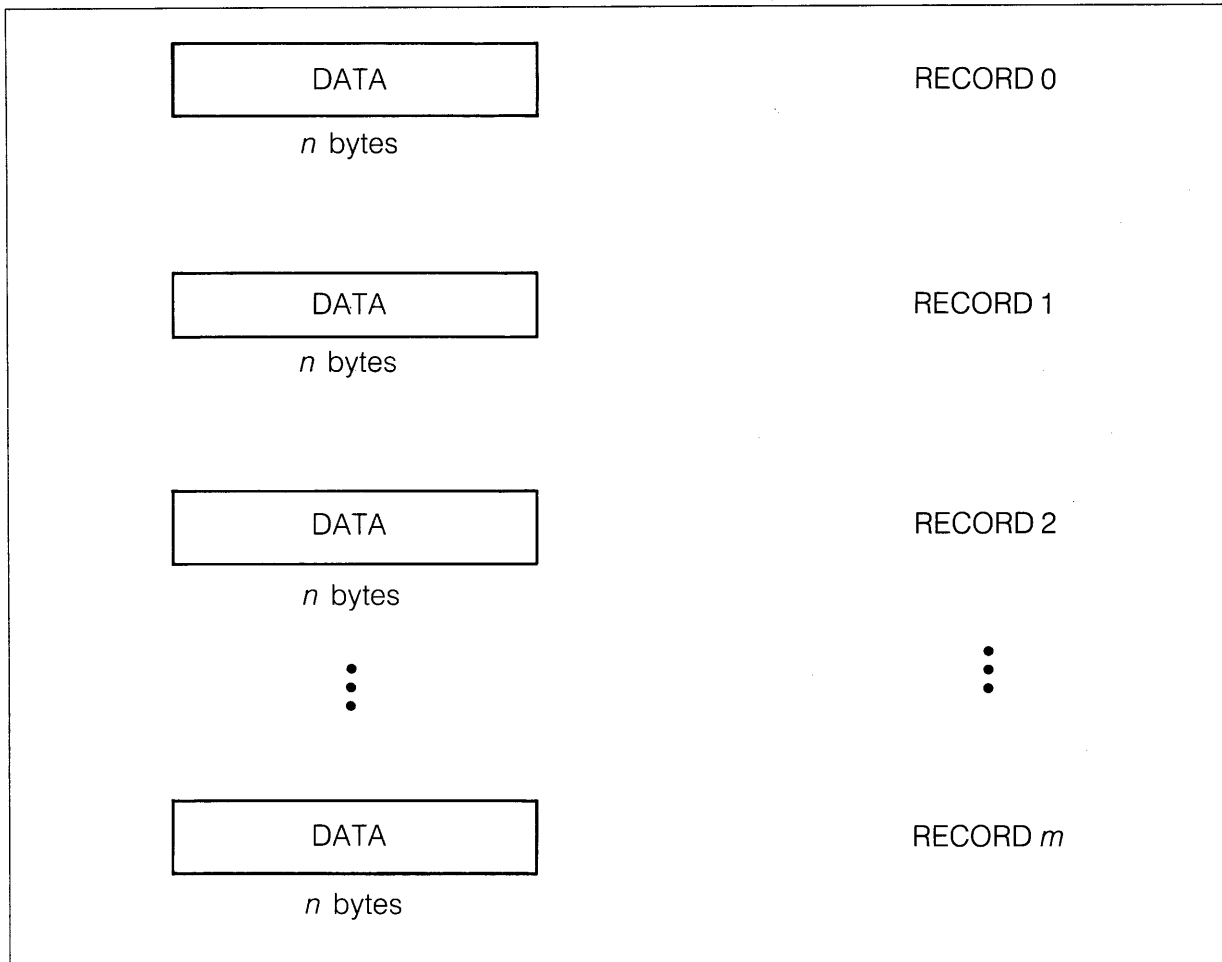
Figure 2-1. Fixed-length records.

Variable-Length Records

There may be a time when you want a disc file in which the logical records need not be the same size. In this case, you can request that the format of the records be **variable-length**. The file system will know the size of each logical record because each record is preceded by a one-word (16-bit) counter giving the length of the record in bytes. Thus, the data for each record is accompanied by an indication of its length. When you build a file containing variable-length records, specify a record size at least large enough to accommodate your longest record.

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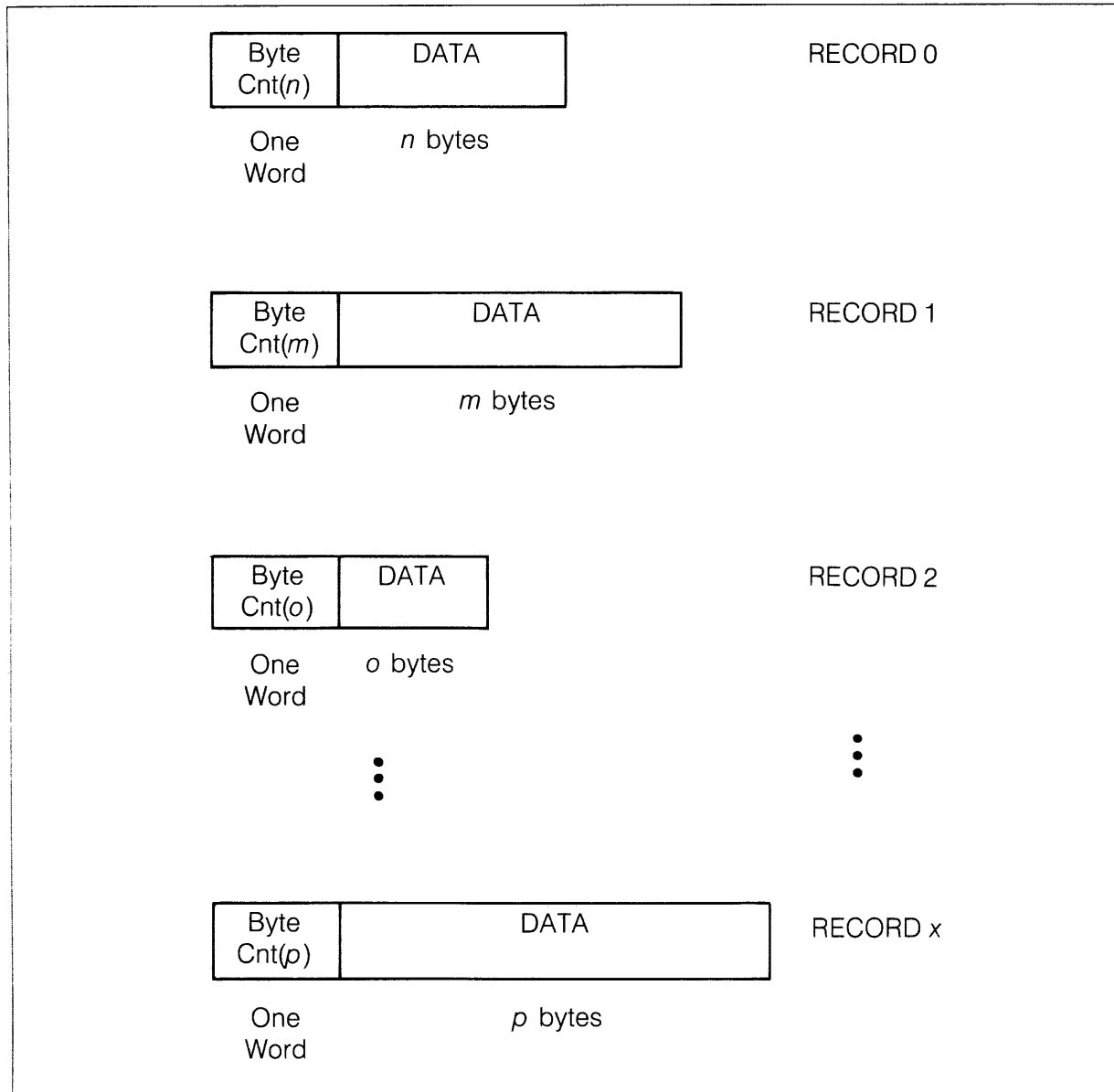
Figure 2-2.   Variable-length records.

Undefined-Length Records

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

The file system knows the maximum room available in each record of a disc file because the same amount of space is allocated for each record; however, the data in the records may vary in length, so some of the space may contain "filler" instead of good data. The file system (and, at times, the I/O system) supplies this "filler" when the length of the data being written is less than the maximum record length. You must know how much of the data in a record is valid, since the file system cannot distinguish good data from filler.

Figure 2-3 depicts a disc file with undefined-length records. When data does not fill the space allocated, filler occupies the unused space.
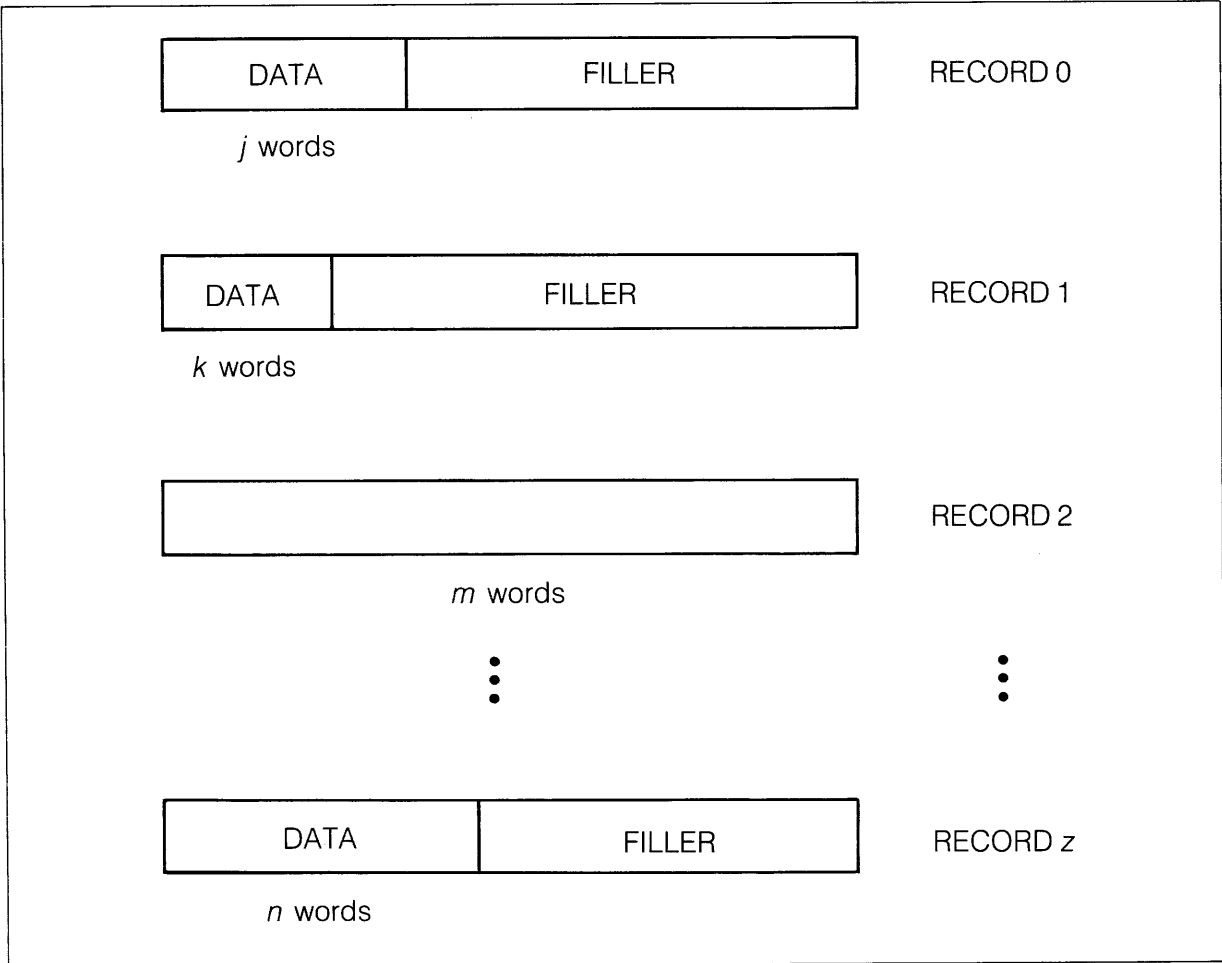
Figure 2-3. Undefined-length records.

NOTE

*The last two bytes of the record will be repeated as filler only to the
end of the sector. Any remaining sectors of record space will be set
to blanks (for ASCII files) or zeroes (for binary files).*

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 disc files) or :FILE
commands, or the FOPEN intrinsic. The file system uses the convention that a negative record size
means the size is 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 essentially a word-oriented machine, so records always begin on a word boundary. This has a particularly important effect on disc and magnetic tape files: odd byte record lengths are always rounded up 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 is rounded up and is accessible for data. However, if the file is ASCII and has fixed- or undefined-length records, the extra byte is not accessible for data. The odd byte length remains the maximum size allowed for data. Figure 2-4 illustrates the placement of odd-byte records and the disposition of the added byte.
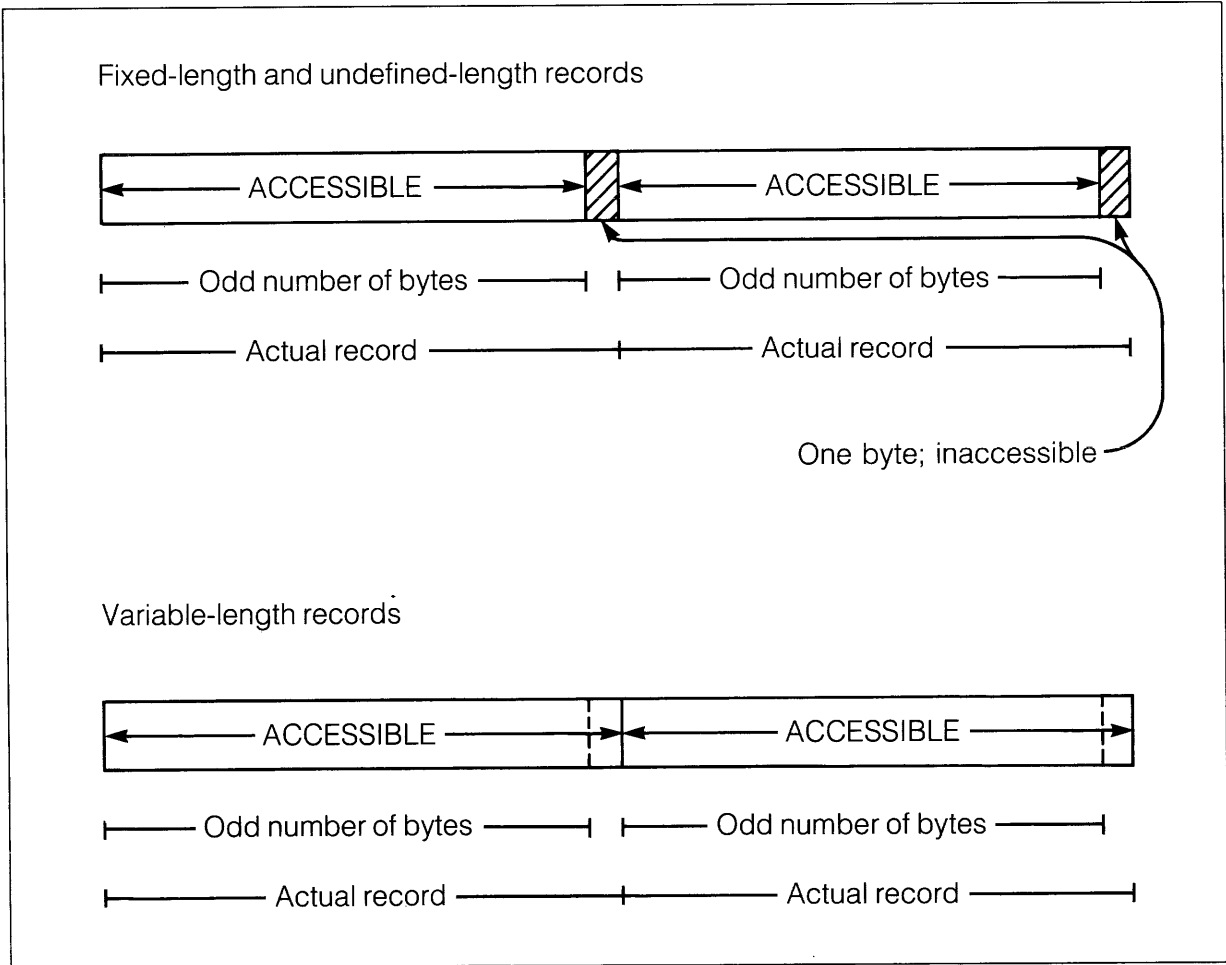
Figure 2-4.   Record placement for ASCII files.

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 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)
Disc	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 disc or tape when you are operating in unbuffered or NOBUF mode. (Buffering is discussed in a later section.) For files on other devices, physical records are determined by 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 MPE :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 disc accesses.

- Better disc space utilization.

NOTE

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

Disc access considerations. Since one disc read or write will transfer one physical record, you can minimize input/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 multiprogramming environments, blocking helps to minimize device head contention on system domain discs and shared private volumes: since one disc access will read or write as many records as one block contains, you and other system users will need to gain control of the disc less often than if you had to read each logical record individually.

NOTE

The blocksize of your file is determined by multiplying two parameters you supply when you create the file: the recsize and the blockfactor. For buffered access, the maximum blocksize allowed is 14K (14,336) words.

Disc space considerations. Discs on the HP 3000 are not word addressable: disc 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 disc space will be wasted.

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

300 words | 84 words

| BLOCK | | WASTED |

128 | 128 | 128
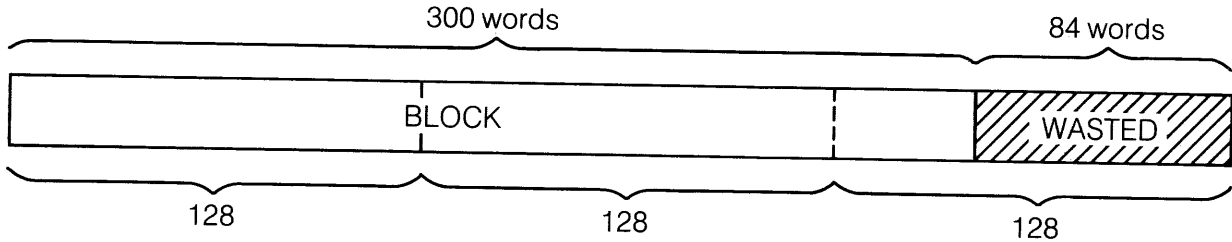
Since each block starts on a sector boundary, the unused space following each block is wasted. If the block size had been 384 words rather than 300, all of the space in three sectors could be used for each block, and no space would be wasted.

Recommendation: To minimize wastage of 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 your 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 *(blockfactor)*. When written to disc, each block begins at the start of a sector and may occupy one or more contiguous sectors. This occurs because the disc controller can only transfer data in units equivalent to the length of a sector, 128 words or 256 bytes. Thus, on a disc file, a 240-byte block containing three 80-byte fixed-length records would appear as follows:

BLOCK

(blockfactor × recsize)

| Logical Record 0 | Logical Record 1 | Logical Record 2 | |

recsize = −80 | recsize = −80 | recsize = −80

240 bytes

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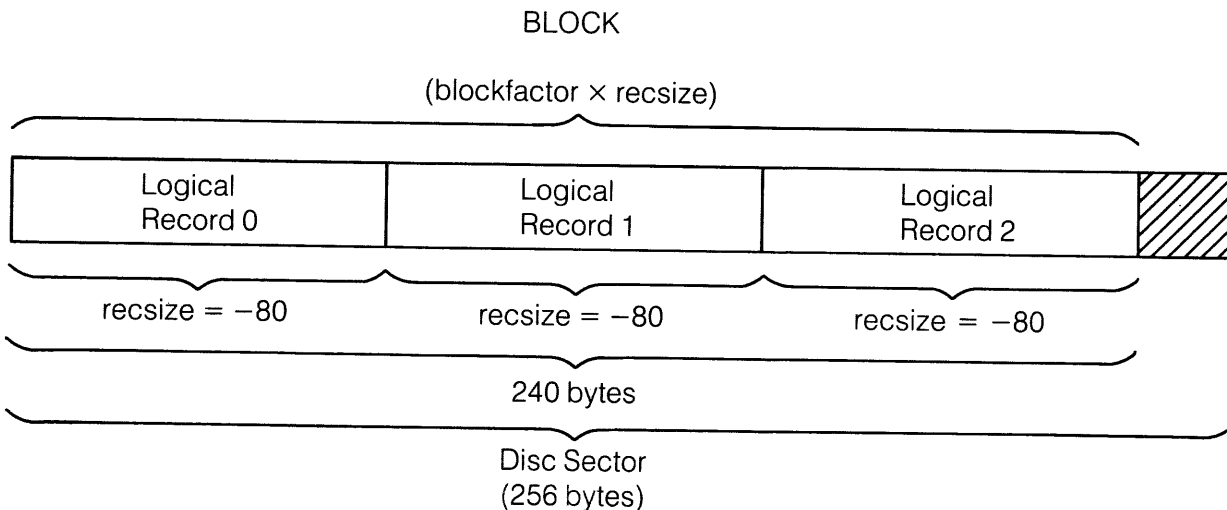
The 16 bytes remaining at the end of the sector are wasted, since this block cannot use them and the next block begins at the start of the next sector. Because of this fact, you can waste disc space if you do not block your records carefully. For example, if you use a blocking factor of 1 when writing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

Block

Sector   Sector

| Logical Record 0 | //////////////////////////// |

256 bytes   2 bytes   254 bytes unused
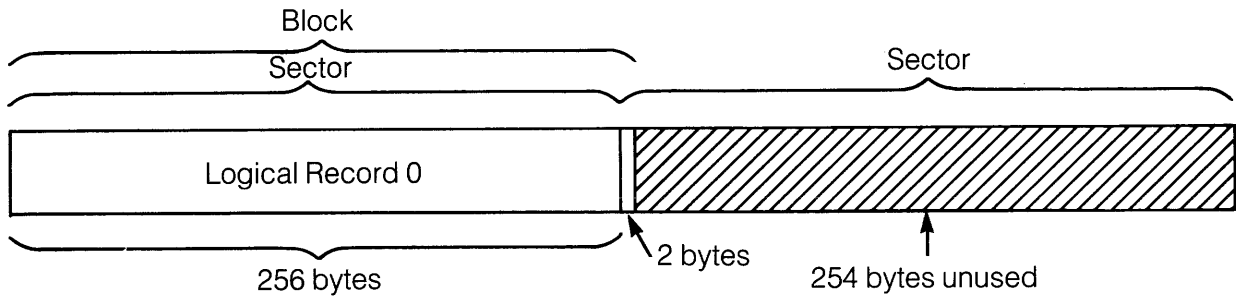
In a large file, this much waste (about 49.6%) soon becomes devastating.

For optimum use of disc space, compute the block size so that:

recsize $\times$ *blockfactor* = a multiple of 256, for bytes

or

recsize $\times$ *blockfactor* = a multiple of 128, for words.

If you can't make the blocks fit into sectors exactly, it is better to have blocks that are 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.

Block

Sector   Sector

| Logical Record 0 | Logical Record 1 | Logical Record 2 | Logical Record 3 | Logical Record 4 | /// |

510 bytes   2 bytes unused

512 bytes
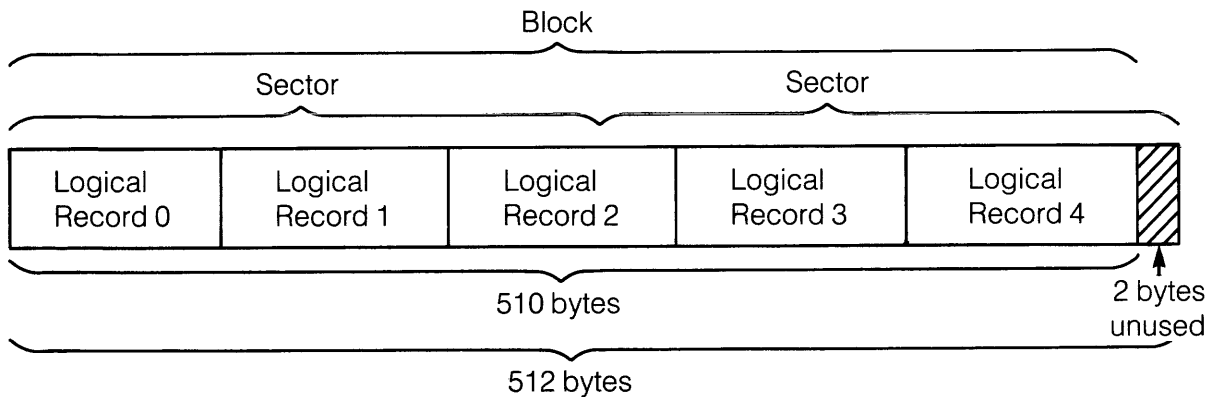
Blocks Containing Variable-Length Records

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

The file system will change the recsize and blockfactor that you specify for your blocks. Your recsize and blockfactor will be multiplied to yield a new recsize, and your blockfactor will be changed to 1. For example, if you request a recsize of 20 words and a blockfactor of 6, your new recsize will be 120 words and your blockfactor will be 1. In both cases, the available space in your blocks is the same: 120 words. The recsize and blockfactor 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 blocksize, in words, will be determined by the formula

$$(recsize + 1) \times blockfactor + 1$$

To avoid permanent waste of file space, make your blocksize 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, disc 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.

REC 1 COUNT	Record 1	REC 2 COUNT	Record 2	−1	////////
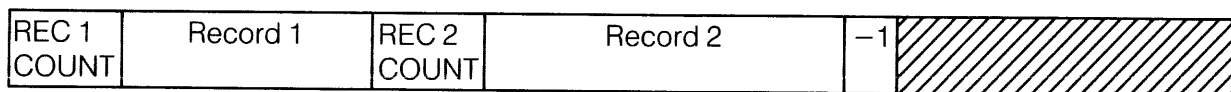
BLOCK

Other records of the same file may fit better:

REC 3 COUNT	Record 3	REC 4 COUNT	Record 4	REC 5 COUNT	Record 5	−1
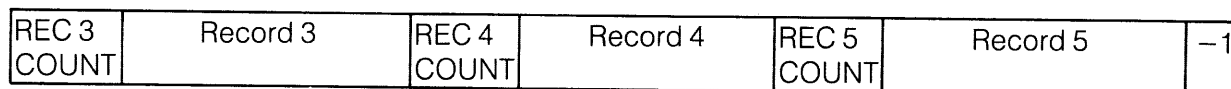
BLOCK

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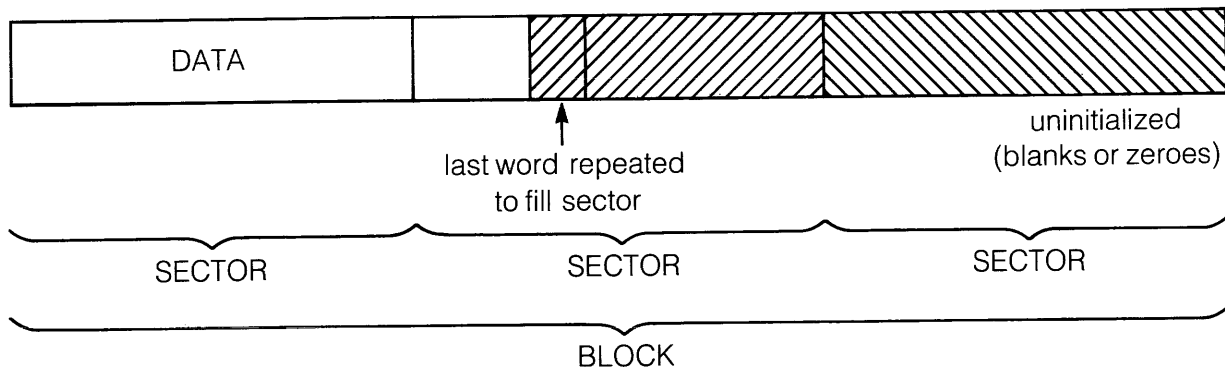
A block size of 384 words (768 bytes) has been specified; each block covers three sectors. The record in this case is only 140 words long. It fills one sector, and ends in the next; the last word of the record is repeated to fill the second sector. The third sector is not used at all, and is filled with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

Blocking Consideration: System File Label

Every disc file has a system file label, which identifies the file to the system and contains the characteristics of the file; the contents of the system file label will be discussed in the section on File Structure. 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 is 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 wastage in this case includes the 35 words left over by the records plus the 512 words lost on the system file label, for a total of 547 words wasted. This situation is illustrated below:
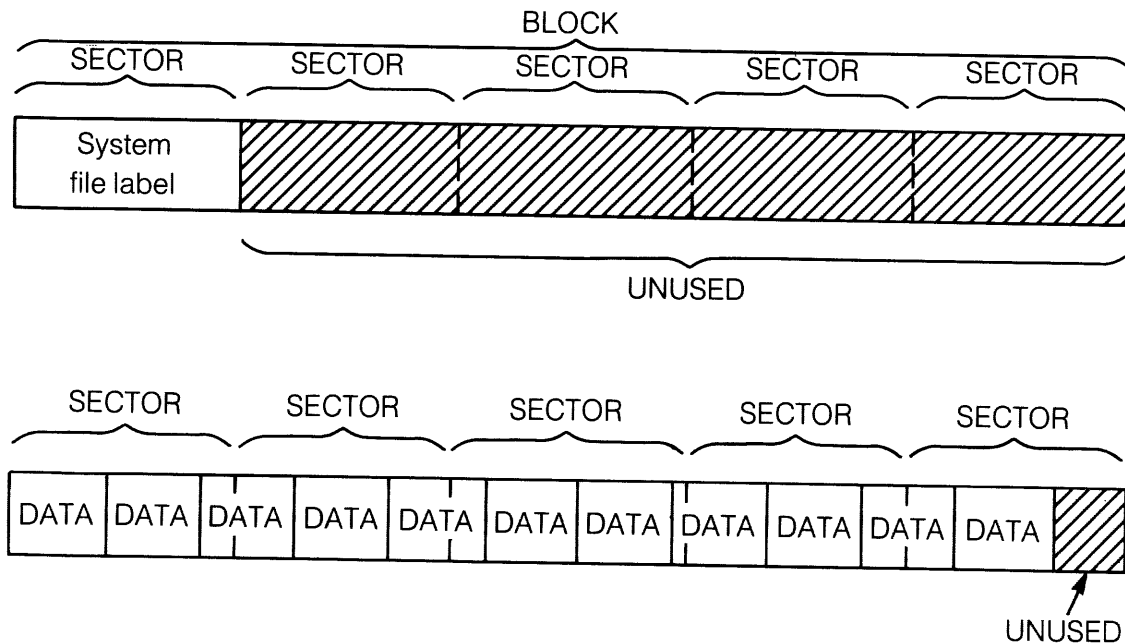
NOTE

The situation just described is rather extreme. Unless your file is very small, it is best to choose your blocking factor in the way that seems most efficient for the records; the space occupied by the system file label is simply a consideration.

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 disc, 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 this value with a minus sign as shown above. When you express record size in words, be sure to omit the minus sign.

In the last section, we discussed the basic units of information available to us: logical records. In this section, 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.
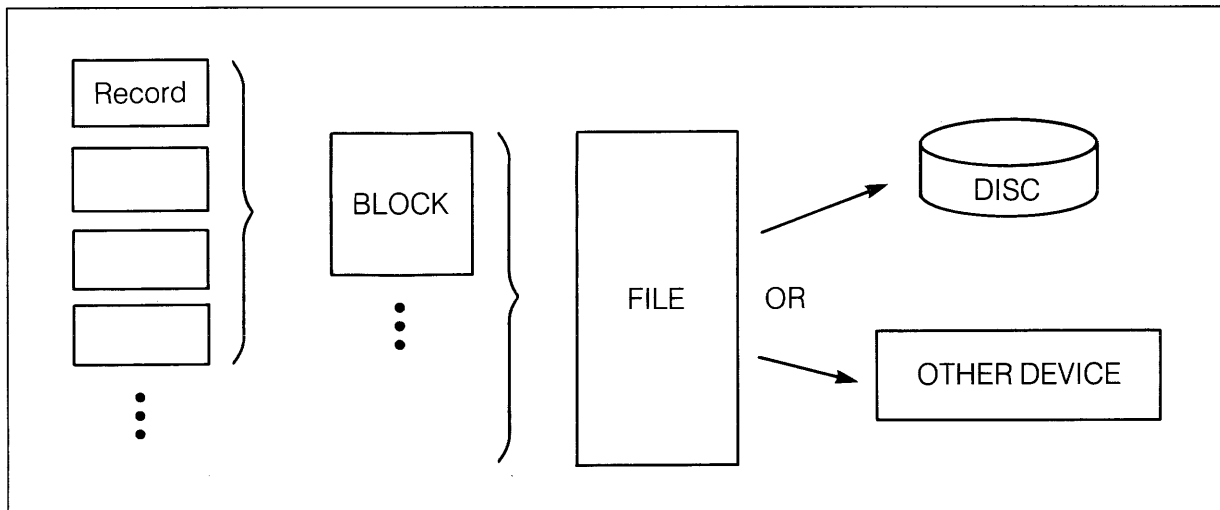
Figure 3-1.   Records/Files Relationship.

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 disc file be distributed on the disc?

●   How should the file be identified?

●   What characteristics should the file have?

DISC FILES AND DEVICEFILES

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

Disc files, which are files residing on disc, immediately accessible by the system and potentially sharable by several sessions/jobs at the same time.

3-1

Devicefiles, which are files currently being input to or output from any peripheral device *except* a disc. (Files on serial disc are considered devicefiles.) 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 devicefile 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 devicefiles, although temporarily residing on disc, are considered devicefiles in the fullest sense because they are always originated on or destined for devices other than disc, and because you generally remain unaware or their storage on disc as an intermediate step in the spooling process. Whether they deal with spooled or unspooled devicefiles, you programs handle input/output as if the files reside on non-disc devices. The Console Operator, not the user, controls the spooling operation.

This section is chiefly concerned with disc files. Devices and devicefiles are discussed at the end of the section.

FILE PLACEMENT

Free space on a disc is 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 disc, or may even be scattered over several discs, 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 re-create 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 disc 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:

$$\text{Extent Size} = \text{Sectors/Extent}$$

$$\text{Sectors/Extent} = \frac{\text{Number of blocks}}{\text{Number of extents}} \times \text{Sectors/Block}$$

$$\text{Sectors/Block} = \frac{\text{Block size (in bytes)}}{256}$$

$$\text{Number of blocks} = \frac{\text{Number of records}}{\text{Block factor}} + 1 \text{ (for file label)}$$

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

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 *blockfactor* of 3. Each record is an 80-byte card image. The extent size is:

$$\text{Number of blocks} = \frac{1024}{3} + 1 = 343$$

$$\text{Sectors/Block} = \frac{240}{256} = 1$$

$$\text{Sectors/Extent} = \frac{343}{8} \times 1 = 43 \times 1 = 43$$

$$\text{Extent Size} = 43 \text{ Sectors/Extent}$$

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

With a *blockfactor* of 16 applied in the above example, the extent size is:

$$\text{Number of blocks} = \frac{1024}{16} + 1 = 64$$

$$\text{Sectors/Block} = \frac{1280}{256} = 5$$

$$\text{Sectors/Extent} = \frac{64}{8} \times 5 = 8 \times 5 = 40$$

$$\text{Extent Size} = 40 \text{ Sectors/Extent}$$

In this case, all extents contain 40 sectors.

NOTE

Spooled devicefiles (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 he configures the system.

The extents that comprise your file will reside on discs 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 disc 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 discs 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 disc specifically because of its speed. In this case, you would reference that disc by its logical device number (ldev #) when you create the file.

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

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 disc 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 disc 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 disc space available to you.

Occasionally, you may wish to override programmatic specifications dealing with extents to optimize the use of disc space. For instance, if your disc 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 disc space you require. To illustrate, if your disc 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 your 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 disc
```

In general, the rule is: if your disc 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 only when it is 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 read 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 it is by building a new file and copying the old one into it.

FOPEN

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

filenum := FOPEN (*formaldesignator, foptions, aoptions,*

recsize, device, formmsg, userlabels,

blockfactor, numbuffers, filesize,

numextents, initalloc, filecode);

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 words 8 extents 1 extent Disc
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 multirec No multiaccess Wait for I/O Numbuf = 2

File domains are discussed in a later section. Settings for the AOPTIONS and FOPTIONS are shown in Appendix A. For more details on using the FOPEN intrinsic, see the MPE Intrinsics Reference Manual, part number 30000-90010.

BUILD

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. Its syntax is:

```
:BUILD filereference  [;DEV =  [ [dsdevice] #] [device] ]
                       [;DISC =  [numrec] [, [numextents]
                                  [,initalloc] ] ]
                       [;REC =  [recsize] [, [blockfactor]
                                 ⎡ ⎡F⎤ ⎡,BINARY⎤ ⎤ ⎤
                                 ⎢,⎢U⎥ ⎢,ASCII  ⎥ ⎥ ⎥
                                 ⎣ ⎣V⎦ ⎣        ⎦ ⎦ ⎦
                       ⎡;CCTL  ⎤
                       ⎣;NOCCTL⎦
                       [;TEMP]
                       [;CODE =  [filecode] ]
                       ⎡;RIO  ⎤
                       ⎣;NORIO⎦
                       ⎡;MSG⎤
                       ⎣;CIR⎦
```

The parameters for BUILD 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 Commands Reference Manual, part number 30000-90009.

FILE

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 BUILD will physically allocate space for a file and define its characteristics, the FILE command may only define how a file will be accessed at run time. A comparison of the parameters for FILE and FOPEN is given in Table 3-2.

Table 3-2.   :FILE vs. 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 (Bits 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 fromat	F V U	Record format foption (Bits 8:2)	Fixed-length records for disc 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 (Bits 12:4)	Read-only access for all devices except output devices, which are assigned output-only access.
Number of buffers	numbuffers NOBUF	numbuffers (Bits 11:5)	2 buffers.
Exclusive/Share access	EXC SEMI SHR	Exclusive access aoption (Bits 8:2)	For read-only access, SHR takes effect; for other modes, EXC.

Table 3.2.   :FILE vs. FOPEN Parameters (Continued).

CHARACTERISTIC	:FILE PARAMETER	FOPEN PARAMETER	MPE DEFAULT
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 programmatically by *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 disc file	(None)	*userlabels*	No user labels processed.
File labels for magnetic tape files	LABEL NOLABEL	Labeled tape foption (Bit 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 disc 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 disc. 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 Commands Reference Manual, part number 30000-90009.

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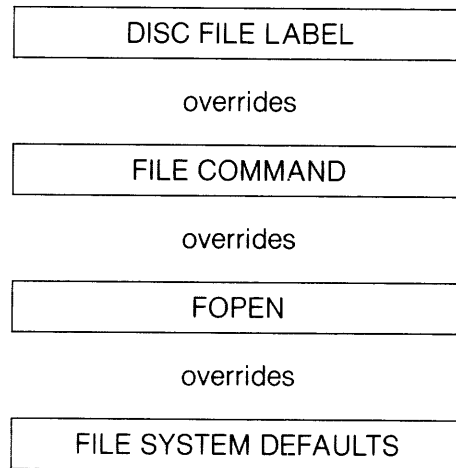
Since the physical characteristics of a file cannot be changed after it has been created, it makes sense that the file label would take precedence over all commands. Other determinants are effective only when a new file is being created.

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, to 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. Here, information about your file's structure, the format of its records, and details about its intended use are permanently recorded; once a file has been created, the system file label cannot be altered.

The contents of a standard disc file label are listed in Table 3-3.

Table 3-3.   Disc 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 0:15)
	(Bit 15:1)
		Not used.
		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) (Bit 1:1) (Bit 2:1) (Bit 3:1)	Store Bit. (If on, :STORE in progress.) Restore Bit. (If on, :RESTORE in progress.) Load Bit. (If on, program file is loaded.) Exclusive Bit. (If on, file is opened with exclusive access.)

Table 3-3.   Disc File Label Contents (Continued).

WORDS		CONTENTS
	(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.
	(Bits 8:8)	Number of user labels.
30-31		Maximum number of logical records.
32-33		Private volume 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.
	(Bits 8:3)	Disc flags.
	(Bits 11:5)	Number of extents, minus 1.
40		Logical size of last block.
41		Extent size.
42-43		Number of logical records in file.
44-107		Two-word addresses of up to 32 disc 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.
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 a files that are frequently updated in order to determine the time of the last update. These special labels can be used for files on disc 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. (For information about tape labels, see Section 9, Magnetic Tape Considerations.)

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 Disc File. When a disc 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. 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.

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.

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. Note that any subsequent FWRITELABEL intrinsic calls will write over an existing label.

```
$CONTROL USLINIT
BEGIN
    BYTE ARRAY DATA1 (0:7): = "DATAONE ";
    BYTE ARRAY DATA2 (0:7): = "DATATWO ";
    ARRAY LABL (0:8): = "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 (Disc).

Reading a User File Label on a Disc 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 the AOPTIONS parameter %4, which specifies input/output access. 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 USLINIT
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 (Disc).

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, upon which they compile object programs. User programs sometimes create files that must be identified in some unique way, too. Such a program might produce a permanent disc file identified by the integer 1. If you were to run this program several times and want to uniquely identify the file produced on each run (or set of runs) by a special class, purpose, or function, you could use a :FILE command to supply a unique file code for each run (or group of runs). For instance, on the second run, you might wish to classify the file with the file code 2, as follows:

```
:FILE 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 prints the file name, file code, and other information about the file. The :LISTF command is discussed in the MPE Commands Reference Manual, part number 30000-90009. Alternatively, you could determine the file code by calling the FGETINFO intrinsic, as discussed in the MPE Intrinsics Reference Manual, part number 30000-90010.

NOTE

For user files, you may use as file codes any number from 0 through 1023. Numbers ranging from 1024 through 1080 are 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.

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

Table 3-4. Reserved File Codes.

Mnemonic	File Code	Meaning
	−400	IMAGE root file.
	−401	IMAGE root file.
	−402	IMAGE file for DS information.
USL	1024	USL file.
BASD	1025	BASIC data file.
BASP	1026	BASIC program file.
BASFP	1027	BASIC fast program file.
RL	1028	RL file.
PROG	1029	Program file.
SL	1031	SL file.
VFORM	1035	V/3000 formsfile.
VFAST	1036	V/3000 fast forms file.
VREF	1037	V/3000 reformat file.
XLSAV	1040	Cross Loader ASCII file (SAVE).
XLBIN	1041	Cross Loader relocated binary file.
XLDSP	1042	Cross Loader ASCII file (DISPLAY).
EDITQ	1050	Edit KEEPQ file (non-COBOL).
EDTCQ	1051	Edit KEEPQ file (COBOL).
EDTCT	1052	Edit TEXT file (COBOL).
	1058	TDP/3000 workfile.
RJEPN	1060	RJE punch file.

Table 3-4. Reserved File Codes (Continued).

Mnemonic	File Code	Meaning
QPROC	1070	QUERY procedure file.
	1071	QUERY work file.
	1072	QUERY work file.
KSAMK	1080	KSAM key file.
GRAPH	1083	GRAPH specification file.
SD	1084	Self-Describing file.
LOG	1090	User Logging logfile.
PCELL	1110	IDS/3000 character set file.
PFORM	1111	IDS/3000 form file.
P2680	1112	IFS/3000 environment file.
OPTLF	1130	On-line performance tool.

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: :FILE commands 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 recognizes your file is its *formal file designator*. This is the file name that is coded into the program, along with the program's specifications for the file. The :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.                                    Block size (1 record per block),
FILE SECTION.                                     intended for card input
FD CARDFILE ←——— File name
    BLOCK CONTAINS 1 RECORDS
    DATA RECORD IS MYDATA
    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 disc file. In such cases, you may also wish to read these records in blocks of three logical records each. Rather than re-code and re-compile the program, you could reference the file in a :FILE command to change the file specifications:

```
Changes file              Formal                  Maintains same record
specifications           designator              ─ size (80 bytes)              Specifies
        \                    /                  /                              ── disc file
         \      :FILE CARDFILE; REC=80, 3; DEV=DISC ◄─────────────────────────
          \    :RUN MYPROG                        ↑
          /                                        \
Runs COBOL program                          Requests block size of
                                              3 logical records
```
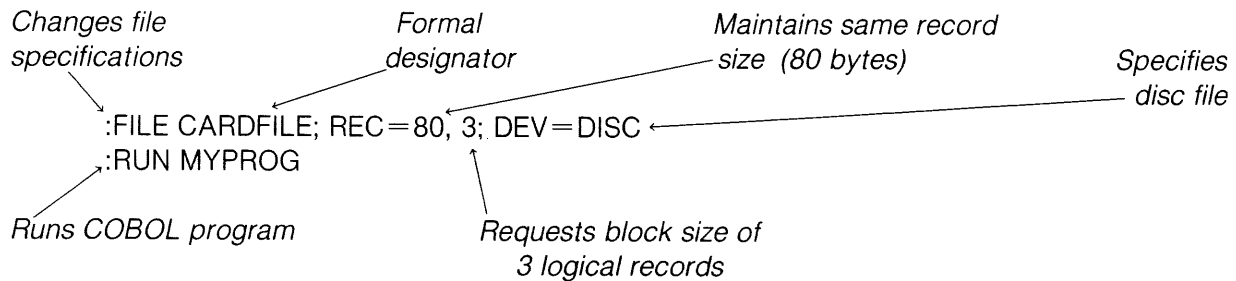
The formal file designator is the name by which your program recognizes the file, but there must also be a means by which the file system can recognize it, allowing it to be referenced by various commands and programs. For an old disc file, this is the file name contained in the file label and in the system or job/session temporary file directory. For a devicefile, it is the name optionally supplied in the :DATA command and copied into the device directory. For a new disc 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 disc or devicefile, this is the "real" name that identifies the file to the file system. It is called the *actual file designator.*

Suppose you create a file and name it DISCFILE; this name is its actual file designator. Suppose, further, that you would like to use DISCFILE in the program MYPROG described above. MYPROG will only recognize a file whose formal designator is CARDFILE, so you use a :FILE command to equate the formal designator to the actual designator:

```
:FILE CARDFILE=DISCFILE
            ↑               ↖
        Formal              Actual
        designator          designator
```
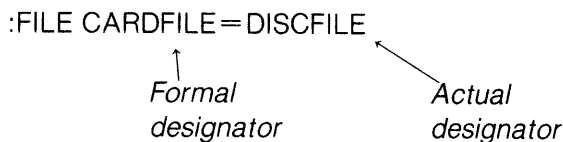
In this way, the :FILE command provides a linkage between the file system's name for a file and your program's name for that file.

Renaming your file. A file's creator can change the file's name, 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 Commands Reference Manual, part number 30000-90009.

DEVICES AND DEVICEFILES

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 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 disc 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 logical device number associated with a device are determined by the System Supervisor or Console Operator when he adds 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	LOGICAL DEVICE NO.	DEVICE CLASSNAME
System Disc (Required)	1	SYSDISC
Disc File	2	DISC
Serial Disc	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 logical device number 5 and device class name CARD. In this case, you could make a unique reference to this device by using either the logical device number 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 logical

devices 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.

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

The :BASIC command is discussed in the MPE Commands Reference Manual, part number 30000-90009.

If a file is a spooled devicefile, 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* when he returns to the machine room. When this is done, your file can be transmitted from disc 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 devicefiles are restricted by the devices on which the files reside. For instance, the file system always assigns a blockfactor of 1 to any file read from a card reader regardless of the blockfactor specified in your FOPEN call or :FILE command. For your convenience, all such device-dependent restrictions are summarized in Table 3-6.

Table 3-6. Device-Dependent Restrictions.

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/OUTPUT 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

---

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 devicefile 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 devicefile 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 devicefile, 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 logical device number for 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 with him 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 devicefile 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 devicefile not requiring them is assigned to the device, the file system automatically asks the operator to mount standard forms or paper.

Foreign Disc Facility

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

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

One way to classify a file is on the basis of its *domain*. A file can be permanent or temporary, or it may exist only to one particular process. The file system maintains separate directories to record the location of temporary (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 this file has not yet been allocated. As a new file, it is known only to the program that creates it, and will exist only while the program is being executed; when the program concludes, the file will simply 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; like a NEW file, it will vanish 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 domains permitted are summarized in Table 4-2.

Table 4-2.   File Domains Permitted.

DEVICE TYPE	DOMAIN
Disc	NEW, OLD, or TEMP
Card Reader	OLD
Paper Tape Reader	OLD
Terminal	NEW or OLD
Printing Reader Punch	NEW or OLD
Synchronous Single-Line Controller	NEW or OLD
Programmable Controller	NEW or OLD
Magnetic Tape Drive	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 can be deleted when it has served its purpose. The disposition parameter of the FCLOSE intrinsic can specify a different domain for a file as it closes, or the :FILE command can be used to change the domain of a file: the DEL, TEMP, and SAVE parameters determine what will happen to the file when it is closed. For details about how the FCLOSE intrinsic handles file domain disposition, see the MPE Intrinsics Reference Manual, part number 30000-90010.

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 purge 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. Suppose 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

The next time it is used, 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 have been programmatically accessed and closed. By using the MPE SAVE command, you can keep a temporary file as permanent without opening and closing the file. Suppose you want to keep a temporary file named TEMPDATA, but do not need to use it in a program at this time. You can enter:

    :SAVE TEMPDATA

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

　　:SAVE $OLDPASS, *filename*

where *filename* is any name you choose. ($OLDPASS and other system-defined files are discussed in a later section.)

For more information about the FILE and SAVE commands, consult the MPE Commands Reference Manual, part number 30000-90009.

DIRECTORY SEARCH

There are two directories with addresses of files: the Job Temporary File Directory 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 Job Temporary File Directory is searched first.

LISTING FILES

To obtain a list of your permanent files, enter the :LISTF command. Use the LISTEQ2 utility to list your temporary files and :FILE equations.The :LISTF command is discussed in detail in the MPE Commands Reference Manual, part number 30000-90009; LISTEQ2 is described in the MPE System Utilities Reference Manual, part number 30000-90044.

FILE OPERATION

In previous sections, we have discussed the structure of files and the records that comprise them. We have learned how to create files that will fill whatever requirements we have. In this section, we will explore the operation and usage of our files. As you read this section, 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, and

System-Defined Files, which the file system defines and makes available to all users to indicate standard input/output devices.

These files are distinguished by the file names and other descriptors (such as group or account names) that reference them, as discussed below. You may use both the file name and descriptors, in combination, as either formal designators within your programs or as actual designators that identify the file to the system. Generally, however, most programmers use only arbitrary names as formal designators, and then equate them to appropriate actual file designators at run time. In such cases, the formal designators (user file names) contain from 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 filereference format, as follows:

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

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

When you reference a file that belongs to your log-on account and group, you need only use the *filereference* 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
        :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 bases for your local file references. Thus, when you log on, if the default file system file security provisions are in effect, you have unlimited access to all files assigned to your log-on group and your home group. Furthermore, you are permitted to read, and execute programs residing in, the Public Group of your log-on 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, always named SYS.

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

```
        :FILE LEDGER=GENACCT.XGROUP ⟵——— Group name
        :RUN MYPROG ⟵——————————————————— Program file (in log-on group)
```

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

    :FILE GENFILE=GENINFO.PUB.SYS

NOTE

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

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

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

You assign a lockword to a new file by specifying it in the *filereference* parameter of the :BUILD command or the *formaldesignator* parameter of the 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 disc.

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

- In batch mode, supply the lockword as part of the file designator (*filereference* format) specified in the :FILE command or 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 imbedded 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 are denied access to the file. In the following example, the old disc file XREF, protected by the lockword OKAY, is referenced:
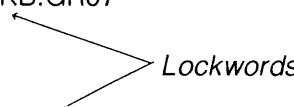
:FILE INPUT=XREF/OKAY ⟵— *Lockword*

- 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 log on. Three examples of :FILE commands referencing lockwords are shown below; the last command illustrates the complete, fully-qualified form of the *filereference* format.

:FILE AFILE=GOFILE/Z22 ⟵— *Lockword*
:FILE BFILE=FILEM/LOCKB.GR07

> *Lockwords*

:FILE CFILE=PAYROLL/X229AD.GROUPN.ACCT0

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 section. 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 re-specify 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 pre-defined files*. Whenever you reference a pre-defined file in a file system command, you must enter the asterisk before the formal designator if you want the pre-definition 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 those 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 disc 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 unneeded output from a running program.

As an example of how to use some of these designators, suppose you are running a program that accepts input from a file programmatically defined as INPUT and directs output to a file programmatically defined as OUTPUT. Your program specifies that these are disc files, but you wish to re-specify 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 Tables 5-2 and 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.
*formaldesignator	Back reference to a previously-defined file.
filereference	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 a successful indication whenever information is written to it.
*formaldesignator	Back reference to a previously-defined file.
filereference	File name, and perhaps account and group names and lockword. Unless you specify otherwise, this is a temporary file residing on disc 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 programmatic requests, with appropriate responses from a person using the input file. For example, an input file and a list file opened to the same teleprinting terminal (for a session) would constitute an interactive pair. An input file and a list file are said to be *duplicative* when input from the former is duplicated automatically on the latter. For example, input from a 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 15.

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

PASSED FILES

Programmers, particularly those writing compilers or other subsystems, sometimes create a temporary disc 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.
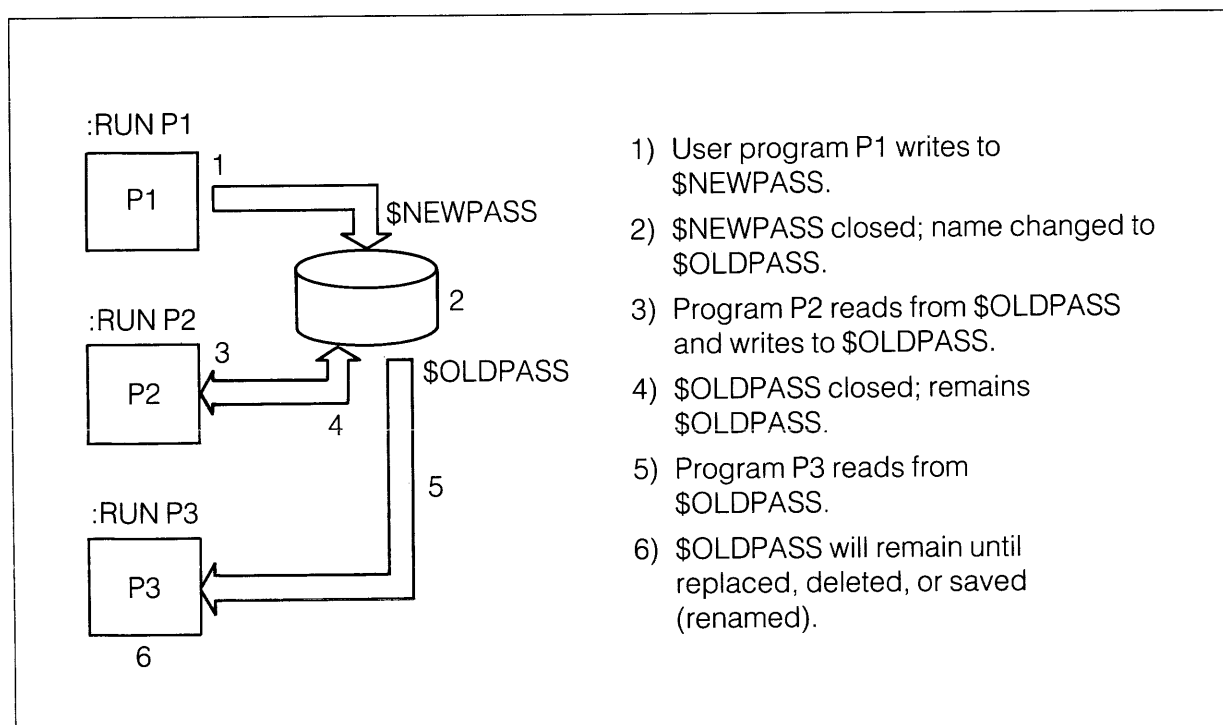
Figure 5-1.   Passing Files between Program Runs

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

```
        •
        •
        •
   :FILE SOURCE1 = DSFIL
   :FILE INTERFIL = $NEWPASS
   :RUN PROG1                        Same file
   :FILE SOURCE2 = $OLDPASS
   :RUN PROG2
        •
        •
        •
```

A program file must pass through several steps as it is executed; passed files are most frequently used between these steps. A program file must be compiled and prepared before it is executed. By default, the compiled form of a textfile is written to $NEWPASS. When the compiler closes $NEWPASS, its name is changed to $OLDPASS; it is this file which 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. This $OLDPASS may be executed any number of times, until it is overwritten by another $OLDPASS file.

The steps that a program takes as it is run are depicted in Figure 5-2.

:SPLGO textfile

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.
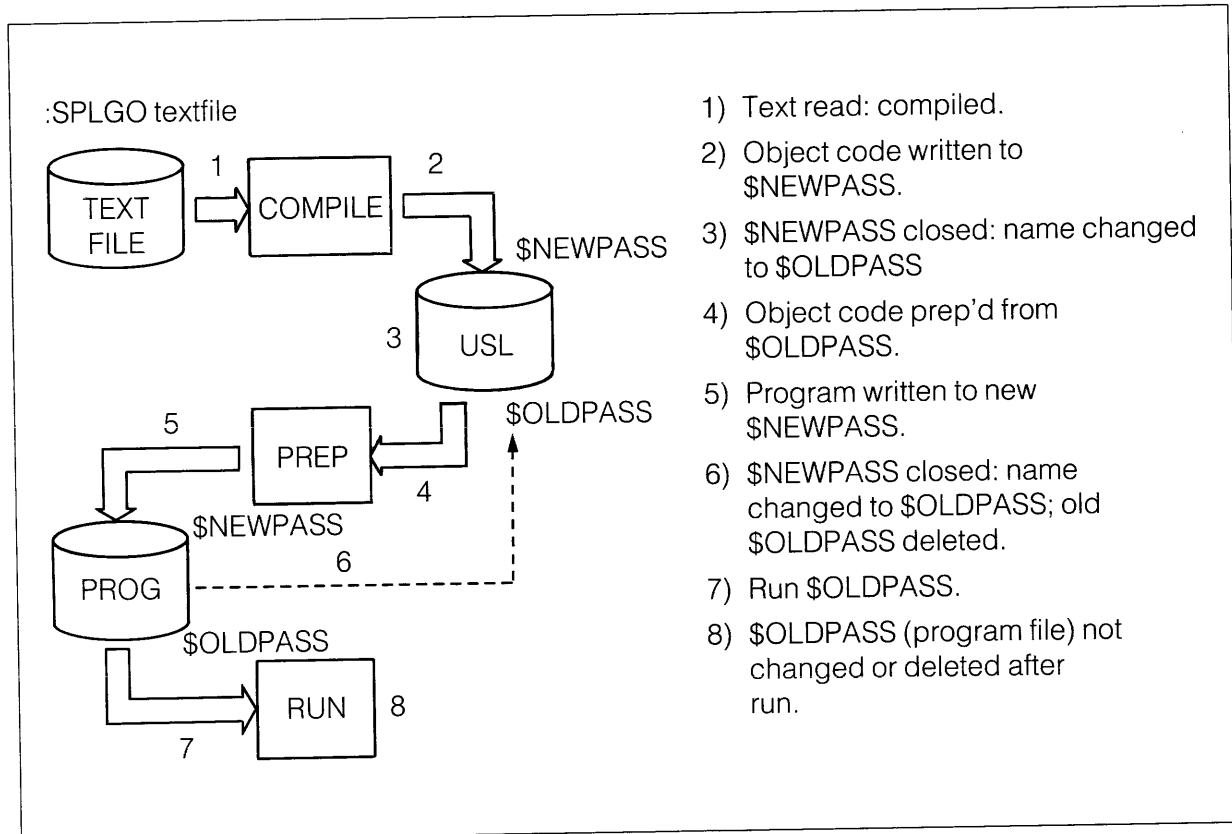
Figure 5-2.   Passing Files within a Program Run

Comparing $NEWPASS and $OLDPASS to Other Disc Files

$NEWPASS and $OLDPASS are specialized disc files with many similarities to other disc files. Comparisons of $NEWPASS to new files, and $OLDPASS to old files, are given in Tables 5-4 and 5-5.

Table 5-4.   New Files vs. $NEWPASS.

NEW	$NEWPASS
Disc space allocated	Disc space allocated
Disc address put into control block	Disc address put into control block

Default close disposition:
    Deallocate space
    Delete control block entry
	Default close disposition:
	    Rename to $OLDPASS
	    Save disc address in job/session table
	        (Job Information Table)
	    Delete control block entry

Disc address not saved
    (Not in any directory)
	Disc address saved for future use in the
	    job/session

Table 5-5.   Old Files vs. $OLDPASS.

OLD	$OLDPASS

Directory (job temporary or system)
    searched for disc address
	Disc address obtained from Job Information
	    Table (JIT)

Disc address put into control block	Disc address put into control block

Default close disposition:
    Delete control block
	Default close disposition:
	    Delete control block

Disc address still in directory for future use	Disc address still in JIT for future use in job/session

You can use the same commands and intrinsics with $NEWPASS and $OLDPASS as you would with any other disc file.

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 MPE, 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 programmatic options through the :FILE command. The options are listed in Table 5-6. The actions taken by MPE 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 other session or job*. Each accessor uses *copy* of portion of file within its own buffer.

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

REQUESTED ACCESS GRANTED, UNLESS NOTED

Requested Access	Current Use	FOPEN for Input		FOPEN for Output		FOPEN for Input/Output	
		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.

Exclusive Access. This option is useful when you wish to update a file, and wish to prevent other users or programs from reading or writing on the file while you are using it. Thus, no user can read information that is about to be changed, nor can he alter that information. To override the programmatic option under which the file would be opened and request exclusive access, you could use the EXC keyword parameter in the :FILE command:

```
:FILE DATALIST;EXC  ◄——— Requests exclusive access
:RUN FLUPDATE
```

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 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 — it not only makes the file available simultaneously to other accessors (in the same job/session), but permits them to use the same buffers, record pointer, 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, multi-access 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.

Note that it is the *first* accessor to a file that sets the allowable access to a file. For example, if the first accessor specifies share access, that is the access that will be allowed to all future accessors. However, if a subsequent accessor specifies an access option that is more restrictive than the first opener's access option, it will remain in effect until the user that requested it closes the file.

The rule for file access restrictions is: the most restrictive access option used is the one that applies.

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, and a typical block 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 disc because the end-of-file pointer is not kept in the file but is kept in core where it can be updated quickly as writes occur. The necessary interlocking is provided by the intrinsics FLOCK and FUNLOCK, which use a Resource Identification Number (RIN) as a flag to interlock multiple accessors.

In the simple case of a file shared between a writer process and a reader process, where the writer is merely adding records to the file, the writer calls FLOCK prior to writing each record and FUNLOCK after writing. The reader calls FLOCK prior to reading each record, and FUNLOCK after reading. If the writing process should execute while the reader is in the middle of a read, the writer will be impeded on its call to FLOCK until the reader signals that it is done by calling FUNLOCK.

Similarly, if the reader should execute while the writer is performing a write, the reader will be impeded on its call to FLOCK until the writer calls FUNLOCK. FUNLOCK ensures that all buffers are posted on the disc 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.This procedure can also be used if the records are to be written in different files; one of the files is used as a "sentinel" file and the processes lock and unlock this file as required.

For more information about the FLOCK and FUNLOCK intrinsics, consult the MPE Intrinsics Reference Manual, part number 30000-90010.

DATA TRANSFER

The chief activities of the file system involve the transfer of data. In this section we will examine how this is accomplished. As you read this section, 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 disc; *logical record pointers* will 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. Buffered files are discussed later in this section.)

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.
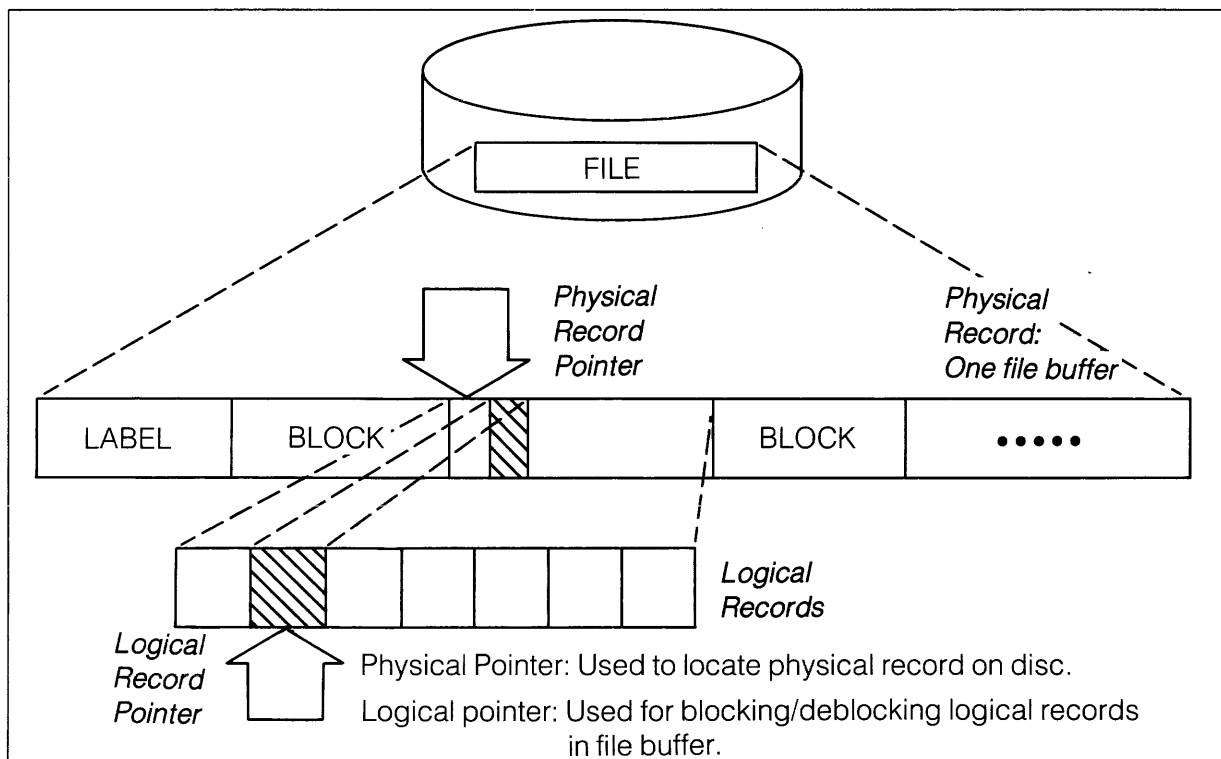
Figure 6-1.   Record Pointers.

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 a 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 a 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. APPEND and other access types will be discussed later in this section.

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

RECORD SELECTION

How are records selected for transfer? 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 indicates the specific location where a file will be accessed; records can be transfered 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.

You may use the default method of record selection with buffered files only.

Random Access

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

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

NOTE

The FREADDIR and FWRITEDIR intrinsics operate in the usual manner to access foreign discs. 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 disc 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 0D (double integers are indicated by the suffix D in SPL) was assigned to REC in statement number 9, 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, etc. The record pointer is advanced by one each time the FREADDIR intrinsic is executed. Since the record number to be read is specified by REC, however, 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 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   TJE, OCT  7, 1975, 10:34 AM

00001000    00000 0     $CONTROL USLINIT
00002000    00000 0     BEGIN
00003000    00000 1       BYTE ARRAY DATA2(0:7):="DATATWO ";
00004000    00005 1       BYTE ARRAY LISTFILE(0:8):="LISTFILE ";
00005000    00006 1       BYTE ARRAY ALTNAME(0:7):="ALTDATA ";
00006000    00005 1       ARRAY BUFFER(0:127);
00007000    00005 1       ARRAY MESSAGE(0:18):="DUPLICATE FILE NAME - FIX DURING BREAK";
00008000    00023 1       INTEGER DFILE2,LIST,ERROR;
00009000    00023 1       DOUBLE REC:=0D;
00010000    00023 1
00011000    00023 1       INTRINSIC FOPEN,FREADLABEL,FREADDIR,FWRITE,FCLOSE,FRENAME,
00012000    00023 1                 FREADSEEK,CAUSEBREAK,FCHECK,PRINT'FILE'INFO,QUIT;
00013000    00023 1
00014000    00023 1       PROCEDURE FILERROR(FILENO,QUITNO);
00015000    00000 1         VALUE QUITNO;
00016000    00000 1         INTEGER FILENO,QUITNO;
00017000    00000 1         BEGIN
00018000    00000 2           PRINT'FILE'INFO(QUITNO);
00019000    00002 2           QUIT(QUITNO);
00020000    00004 2         END;
00021000    00000 1
00022000    00000 1       <<END OF DECLARATIONS>>
00023000    00000 1
00024000    00000 1           DFILE2:=FOPEN(DATA2,%6,%4,128);       <<OLD TEMP FILE>>
00025000    00011 1           IF < THEN FILERROR(DFILE2,1);         <<CHECK FOR ERROR>>
00026000    00015 1
00027000    00015 1           LIST:=FOPEN(LISTFILE,%14,%1);         <<$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(LIST,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(DFILE2,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>>
00041000    00072 1           FREADSEEK(DFILE2,REC);                <<FILL SYSTEM BUFFER>>
00042000    00075 1           IF < THEN FILERROR(DFILE2,6);         <<CHECK FOR ERROR>>
00043000    00101 1
00044000    00101 1           FWRITE(LIST,BUFFER,35,0);             <<ALTERNATE RECORDS>>
00045000    00106 1           IF <> THEN FILERROR(LIST,7);          <<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 2               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(LIST,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.

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 process.

If a condition code of CCE is returned, the

IF <> THEN FILERROR (DFILE2,6);

statement 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.

Data Transfer

```
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
00003000  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,FCLOSE,
00011000  00011 1               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(DFILE1,,,,,,,,,,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 REDC NO>>
00041000  00076 1         FWRITEDIR(DFILE2,BUFFER,128,REC);   <<INVERT REC ORDER>>
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.   FWRITEDIR Example.

Optimizing Direct-Access File Reading

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

The FREADSEEK intrinsic moves the record from the file tò 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 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 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 disc file, you use the FUPDATE intrinsic. FUPDATE affects the *last* logical record (or block for NOBUF files, to be discussed later) accessed by any intrinsic call for the file named, and writes information from a buffer in the stack into this record.Following the update operation, the record pointer is set to indicate the next record position. The record number need not be supplied in the FUPDATE intrinsic call; FUPDATE automatically updates the last record referenced in any intrinsic call. Note that the file system assumes the record to be updated has just been accessed in some way.

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 disc file. Figure 6-4 contains a program that opens an old disc 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 30. 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.

```
PAGE 0001   HEWLETT-PACKARD 32100A.05.1   SPL/3000   TJE, OCT  7, 1975, 10:32 AM


00001000   00000 0   $CONTROL USLINIT
00002000   00000 0   BEGIN
00003000   00000 1     BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000   00005 1     ARRAY BUFFER(0:127);
00005000   00005 1     INTEGER DFILE1,LGTH,DUMMY,IN,LIST;
00006000   00005 1
00007000   00005 1     INTRINSIC FOPEN,FREAD,FUPDATE,FLOCK,FUNLOCK,FCLOSE,
00008000   00005 1               PRINT'FILE'INFO,QUIT,FWRITE,FREAD;
00009000   00005 1
00010000   00005 1     PROCEDURE FILERROR(FILENO,QUITNO);
00011000   00000 1       VALUE QUITNO;
00012000   00000 1       INTEGER FILENO,QUITNO;
00013000   00000 1       BEGIN
00014000   00000 2         PRINT'FILE'INFO(FILENO);
00015000   00002 2         QUIT(QUITNO);
00016000   00004 2       END;
00017000   00000 1
00018000   00000 1     <<END OF DECLARATIONS>>
00019000   00000 1
00020000   00000 1         DFILE1:=FOPEN(DATA1,%5,%345,128);     <<OLD DISC FILE>>
00021000   00011 1         IF < THEN FILERROR(DFILE1,1);         <<CHECK FOR ERROR>>
00022000   00015 1
00023000   00015 1         IN:=FOPEN(,%244);                     <<$STDIN>>
00024000   00024 1         IF < THEN FILERROR(IN,2);             <<CHECK FOR ERROR>>
00025000   00030 1
00026000   00030 1         LIST:=FOPEN(,%614,%1);                <<$STDLIST>>
00027000   00040 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 RECD>>
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(30),5);        <<EMPLOYEE NUMBER>>
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.   FUPDATE Example.

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

Table 6-1. Intrinsics for Data Transfer.

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

NOTE

*The access modes mentioned in Table 6-1 are discussed in the
section on File Security.*

Relative I/O

In addition to the conventional random and serial access, MPE offers Relative I/O access. 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 to you; however, they can be read by random access using FREADDIR. They may be overwritten both serially and randomly using FWRITE, FWRITEDIR or FUPDATE. With RIO access the internal structure of RIO blocks is transparent.

CONTROL OPERATIONS

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

Spacing: Move the record pointer backward or forward.

Pointing: Set the record pointer to a particular value.

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. (Append and other access types are discussed later in this section.) Attempting to space 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. Its syntax is

FPOINT *(filenum,recnum)* ;

Use the recnum parameter to specify the new location for the record pointer: recnum is the record number relative to the start of the file (record 0).

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. (Append and other access types are discussed later in this section.) 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. FCONTROL's 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. (Access modes will be discussed in the section on File Security.)

NOTE

*FCONTROL 5 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 Intrinsics Reference Manual, part number 30000-90010.

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

Old group   *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 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. Log on 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 log on 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:

Old account name

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

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 log on 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 disc in a special format, transporting that tape or disc 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 FILEZ;DEV = TAPE
:RESTORE *FILEZ

You can also transfer files by copying them to magnetic tape or serial disc via FCOPY and transporting that tape or disc pack to the new system and loading it. The method for doing this is discussed in the FCOPY Reference Manual, part number 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.
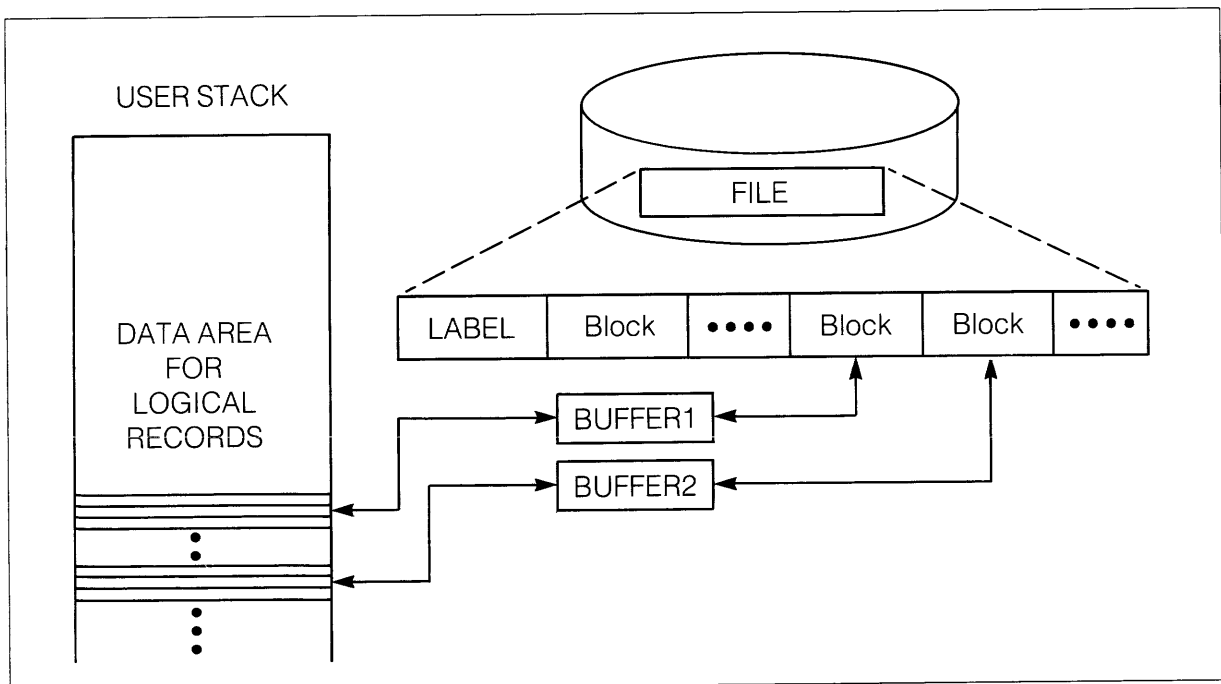
Figure 6-5.   Data Transfers using Buffers.

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 block 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 illustrates 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.
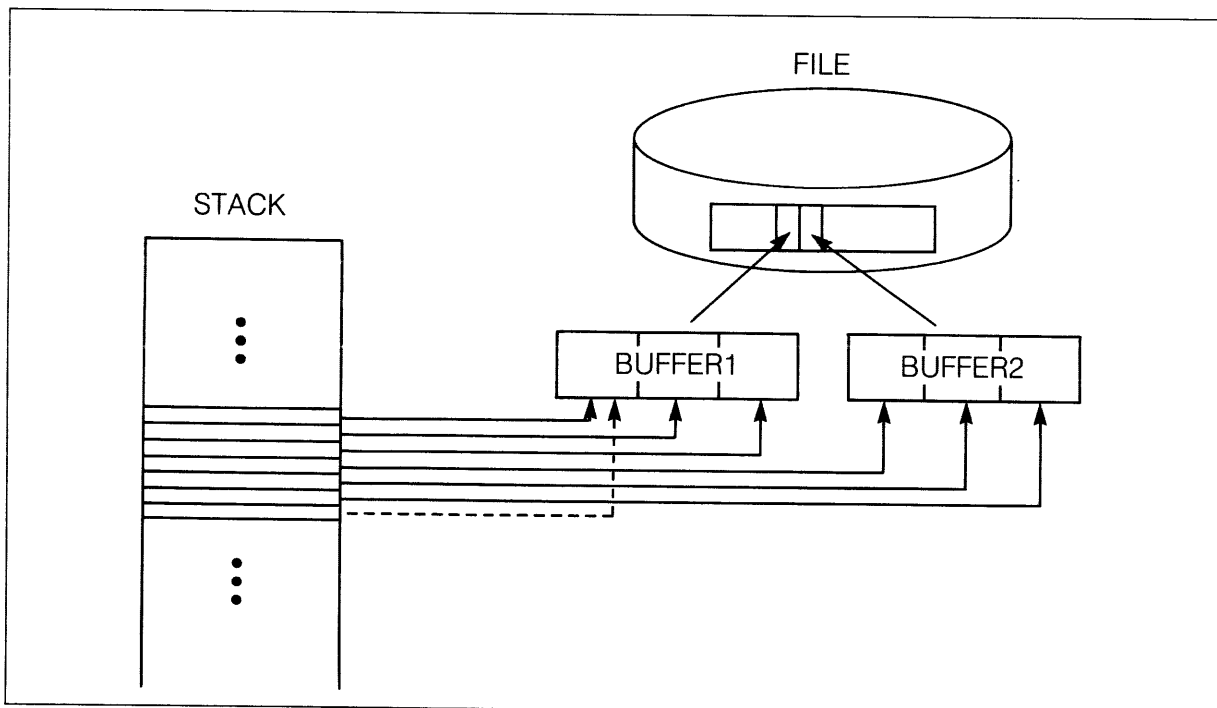
Figure 6-6. Buffer Operation.

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 needlessly occupies space in main memory. If you request 0 buffers, the file system will override this and supply the standard default of 2.

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 14K words (14,336 words). This means that if a file has one buffer, that buffer may be up to 14K words in size; if a file has two buffers, they may each be up to 7K words in size, and so on. If the total buffer size you request is too large (that is, blocksize X numbuffers > 14K) 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).

Why Buffer Transfers?

There are two major advantages to buffering data transfers: buffering results in automatic blocking and deblocking of logical records and 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 disc 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 non-sequentially.

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, discussed later in this section, 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 BIGDATA;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 containing variable-length records.

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*
         ↓
    :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. IOWAIT is discussed in the MPE Intrinsics Reference Manual, part number 30000-90010.

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

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 data from a disc 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; it will not, however, break up blocks or sectors; your transfers 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 Intrinsics Reference Manual, part number 30000-90010. The transfer length will be returned in the TLOG parameter of FCHECK.

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 in the NOBUF case.

When you issue an FCONTROL intrinsic call with a controlcode 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 controlcode 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 in the file label; this will protect data from being lost in the case of a system crash.

The FSETMODE and FCONTROL intrinsics are discussed in detail in the MPE Intrinsics Reference Manual, part number 30000-90010.

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 or group, or to that particular file. These restrictions are based on three factors:

Modes of access (reading, writing, or saving, for example.)

Types of users (users with Account Librarian or Group Librarian capability, or creating users, for example) to whom the access modes specified are permitted.

Use of private volumes. Allows users to access files residing on private disc volume sets.

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

SPECIFYING AND RESTRICTING FILE ACCESS BY ACCESS MODE

When a program creates a file, it can define the way the file can be accessed by specifying a particular access mode (such as read-only, write-only, update, and so forth) for the file. These specifications apply to files on any device, and can be changed or overridden only by yourself, as the creator of the file. They are discussed in the following paragraphs. In addition, for a file on disc, a program can also restrict access so that only one access-attempt (FOPEN call) or process (running program) can access it at one time, or can allow it to be shared among several accessors.

The access types that can be specified by a program are listed in Table 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 devicefiles such as card reader and paper-tape reader files, as well as magnetic tape, disc, 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 devicefiles such as card punch, line printer, as well as tape, disc, and terminal output files.

Table 7-1.   File Access Mode Types (Continued).

ACCESS MODE	:FILE PARAMETER	DESCRIPTION
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 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 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 for a sample file. 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 another record were appended to the file, it would be the eleventh record. When you open the file in the write-only mode, however, all records presently in the file are deleted and the logical record pointer and current EOF move to Record 0. Now when you write a record to the file, this will be the first record in that file.

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 programmatic specifications by using the :FILE command to request append access to the file, as follows:

```
:FILE TASK; DEV=TAPE; ACC=APPEND
:RUN PROGM
```

Requests append access

Suppose you run a program that opens a disc 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 also permits the read, write, and append access modes. 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
:RUN PROGN
```

Requests update access

7-3

Consider a program that reads input from a terminal (file name INDEV) and directs output to a line printer (OUTDEV). You can re-direct 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 OF USER

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 or Account Manager capability bypass the standard security mechanism. A System Manager has unlimited file access to any file in the system, but can save files only in his own account; an Account Manager user has unlimited access to any file within the account. 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 he creates the account. The initial provisions can be changed at any time, but only by that user. At the account level, five access modes are recognized:

    Reading (R)
    Appending (A)
    Writing (W)
    Locking (L)
    Executing (X)

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

    Any User (ANY)
    Account Member (AC)

If no security provisions are explicitly specified for the account, the following provisions are assigned by default:

- For the system account (named SYS), through which the System Manager user initially accesses the system, reading and executing access are permitted to all users; appending, writing, and locking access are limited to account members.

NOTE

Symbolically, these provisions are expressed as follows:

(R,X:ANY;A,W,L:AC)

In this format, colons are interpreted to mean, ". . . is permitted only to . . ." or ". . . is limited to . . .". Commas are used to separate access modes or user types from each other. Semicolons are used to separate entire access mode/user type groups from each other.

- For all other accounts, the reading, appending, writing, locking, and executing access modes are limited to account members (R, A, W, L, X: AC).

GROUP-LEVEL SECURITY

The security provisions that apply to all files within a group are initially set by an Account Manager user when he creates the group. They can be equal to or more restrictive than the provisions specified at the account level. (The group's security provisions also can be less restrictive than those of the account — but this effectively results in *equating* the group restrictions with the account restrictions. A user failing security checking at the account level is denied access at that point, and is not checked at the group level.) The initial group provisions can be changed at any time, but only by an account-managing user for that group's account.

At the group level, six access modes are recognized:

Reading (R)
Appending (A)
Writing (W)
Locking (L)
Executing (X)
Saving (S)

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

Any User (ANY)
Account Librarian User (AL)
Group Librarian User (GL)
Group User (GU)
Account Member (AC)

If no security provisions are explicitly specified, the following provisions apply by default:

- For a public group (named PUB) whose files are normally accessible in some way to all users within the account, reading and executing access are permitted to all users; appending, writing, saving, and locking access are limited to account librarian users and group users (including group librarian users). (R, X: ANY; A, W, L, S: AL, GU).

- For all other groups in the account, reading, appending, writing, saving, locking, and executing access are limited to group users. (R, A, W, L, X, S: GU).

FILE-LEVEL SECURITY

When a file is created, the security provisions that apply to it are the default provisions assigned by MPE at the file level, coupled with the user-specified or default provisions assigned to the account and group to which the file belongs. At any time, however, the creator of the file (and *only* this individual) can change the file-level security provisions, as described in the following pages. Thus, the total security provisions for any file depend upon specifications made at all three levels: the account, group, and file levels. A user must pass tests at all three levels — account, group, and file security, in that order — to successfully access a file in the requested mode.

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

- For all files, reading, appending, writing, locking, and executing access are permitted to all users. (R, A, W, L, X: ANY).

Because the total security for a file always depends on security at all three levels, a file not explicitly protected from a certain access mode at the file level may benefit from the default protection at the group level. For example, the default provisions at the file level allow the file to be read by any user — but the default provisions at the group level allow access only to group users. Thus, the file can be read only by a group user.

In summary, the default security provisions at the account, group, and file levels combine to result in overall default security provisions as listed in Table 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.

- Reading and executing access (only) to all files in the public group of his account and the public group of the System Account.

The important file security rules may be defined as follows:

- Users can create files in their own accounts.

- Only the creator can modify a file's security.

- If a lockword is present on a file, then it is required in order to access the file.

- Account Managers have unlimited access to the files within their accounts.

- System Managers have unlimited access to any file, but can save files only in their account.

Table 7-4. Default Security Provisions.

FILEREFERENCE	FILE	ACCESS PERMITTED	SAVE ACCESS TO GROUP
filename.PUB.SYS	Any file in Public Group of System Account.	(R,X:ANY; W:AL,GU)	AL,GU
filename.group-name.SYS	Any file in any group in System Account.	(R,W,X:GU)	GU
filename.PUB.ac-countname	Any file in Public Group of any account.	(R,X:AC; W:AL,GU)	AL,GU
filename.group-name.accountname	Any file in any group in any account.	(R,W,X:GU)	GU

CHANGING SECURITY PROVISIONS OF DISC 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 disc 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:

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

You always retain the ability to change the security provisions of a file that you have created, even when you are not allowed to access the file in any mode. Thus, you can change the provisions to allow yourself access.

SUSPENDING AND RESTORING SECURITY PROVISIONS

You may temporarily suspend the security restrictions on any disc 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. (File lockword protection, however, is not removed by this command.) The :RELEASE command does not modify the file security settings recorded in the system; it 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

Except for the lockword you removed, the original security restrictions for the file will be in effect.

Interprocess communication (IPC) is a facility of the file system which permits multiple user processes to communicate with one another in an easy and efficient manner. 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 entries made by FWRITEs and deletions made by FREADs: one process may submit records to the file with the FWRITE intrinsic while another process takes records from the file using FREADs.

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 situations, 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 head of the message file and writers access the tail. 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 multirecord parameter is corrected. If FOPEN is used to access a new file, a new message file is created.

NOTE

In bits 12:4 of FOPEN's AOPTIONS, you can specify several different types of writer processes. 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. These AOPTIONs are discussed later in this section.

FREAD   Reads one record from the head 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 head of the file. If a process tries to read from an empty message file which writers are accessing, the file system causes it to wait 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 are no writers, the 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 tail of a message file. If a process tries to write to a full message file which readers are accessing, the file system causes it to wait 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 during 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 section.

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, regardless of what it was previously.

Additional Features

Besides the regular attributes of IPC and message files, several features are available for use with these facilities. Writer ID's, nondestructive reads, and software interrupts are specifically intended for use with IPC; copy access, the global multiaccess option and the ability to append to variable-length record files are general enhancements to the file system. The time-out feature has been expanded to apply to IPC.

Writer ID's. 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. Whenever a writer opens or closes a message file, records are written to the file indicating these actions. 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. A reader may use the writer ID's 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 writen 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.

Global multiaccess. When the global multiaccess option is requested, processes located in different jobs or sessions may open the same file. The global multiaccess option may be requested in the AOPTIONs of the FOPEN to the file, or by using the GMULTI keyword in a :FILE command to define the file.

NOTE

Global multiaccess is unavailable to message files when they have
been opened with exclusive access in copy mode.

Appending to variable-length files. Variable-length files may be opened with append access. It is not necessary to have fixed-length records of the maximum possible size, so space is conserved.

Software interrupts. You may specify that your FREAD and FWRITE completion processing be done with an "interrupt" procedure you supply 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. Software interrupts are discussed in greater detail later in this chapter.

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. Circular files are discussed later in this section.

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 is maintaining 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   Since records are written to the message file every time a writer process opens or closes the file, 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 to you.)

4   Each extent is the same size; that is, 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:

The file system adds two records to allow for one open and one close indication; the number of records goes from 51 to 53.

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.

An additional block is added to the file to accommodate the file label. Now the file contains 19 blocks.

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 that are omitted 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 the 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 zeroes (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; that is, 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) - Multiaccess mode. This feature permits processes located in different jobs
or sessions to open the same file.

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

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.

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:*

0 - read by logical record

1 - read by physical block

Writers must open message files with NOBUF if they are in copy mode; they will
access the file block by block.

(8:2) - Exclusive. The values for this field are the same as for any disc 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) - Multirecord. For message files, the file system sets this bit to 0.

(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 disc; specification of any device other than a disc 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

A few controlcodes deal specifically with IPC. Those not mentioned here are invalid when IPC is being used.

CONTROLCODE	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 time-out 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 time-outs on the file.
6	-	Write End-Of-File. Used only to verify the state of the file by writing out the file label and buffer area to disc; 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.

CONTROLCODE	PARAM	DESCRIPTION
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.)

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 - data 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 FREADs will be unaffected. (Default.)
	FALSE	The next FREAD by this reader will delete the record.
48	—	Arm/disarm 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. If the value of PARAM is 0, the interrupt mechanism is disabled for this file.

FCHECK

There is one 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	*plabel* of the user's soft interrupt procedure. A value of zero implies that soft interrupts are not being used.

Certain intrinsics are not allowed for message files. (The FSETMODE intrinsic is permitted, but ignored.) The disallowed intrinsics are listed in Table 8-1:

Table 8-1.   Intrinsics that are not permitted with message files.

FPOINT	FREADDIR
FREADSEEK	FSPACE
FUPDATE	FWRITEDIR
FDELETE	

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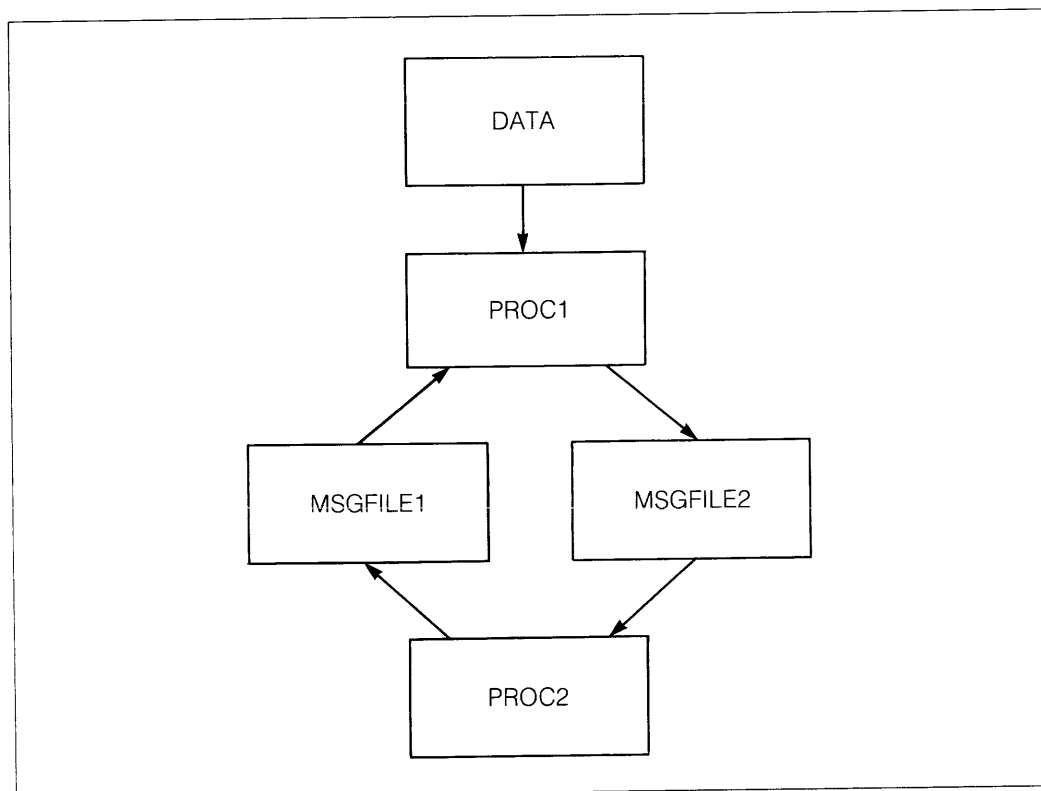
Figure 8-1.   Data Paths among Processes and Message Files.

```
$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 multiaccess 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 multiaccess 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 := 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 (13);          << to finish.                >>

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


$CONTROL USLINIT

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

8-15

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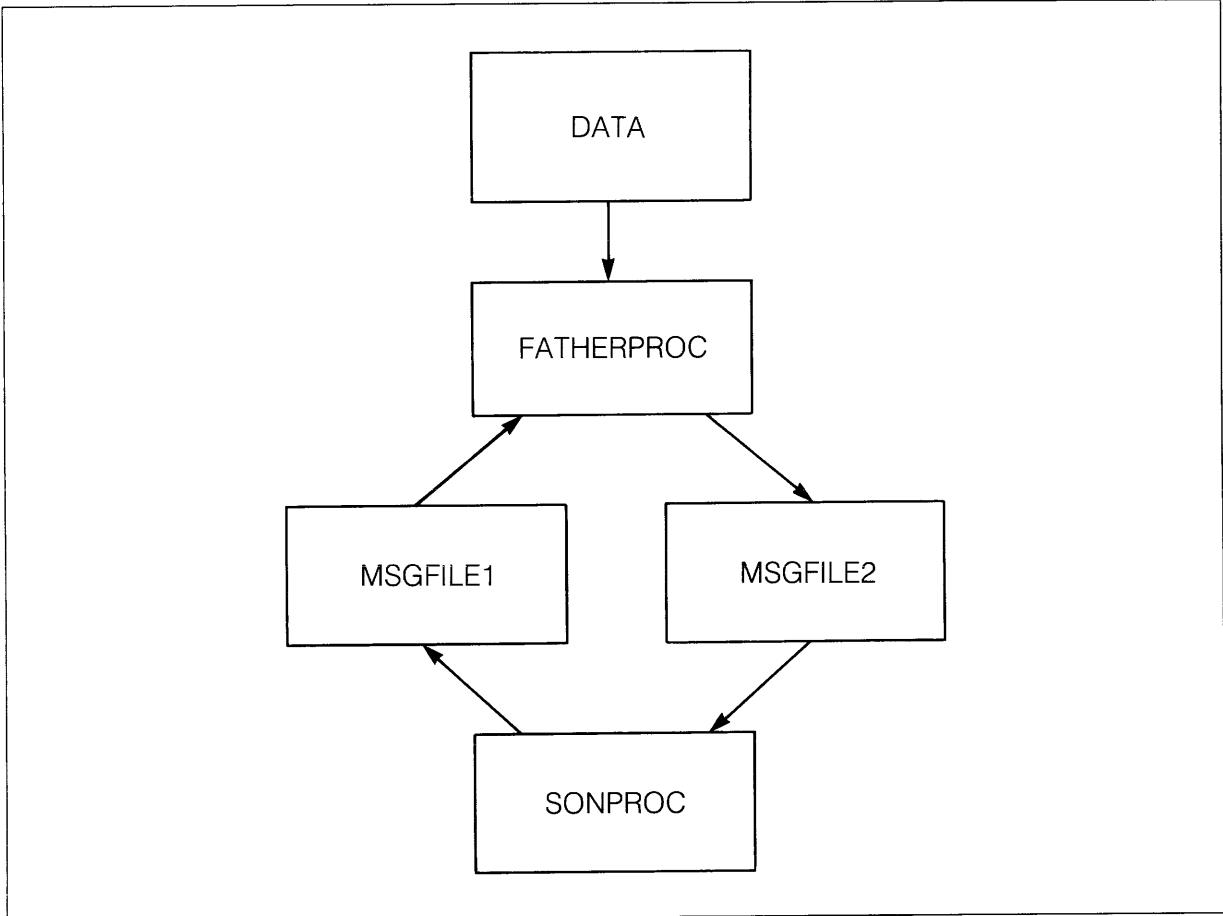
Figure 8-2.   Data Paths among Processes and Message Files.

```
$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.
 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 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 exclu-
* sive access and multiaccess 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 exclu-
* sive access and multiaccess 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 = 0
         %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 multiaccess 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 exclu-
 * sive access and multiaccess 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.
```

SOFTWARE INTERRUPTS

The software interrupt facility enables you to perform FREAD and FWRITE completion process-ing 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:=FINTSTATE(TRUE);
```

The FINTSTATE intrinsic 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 reenable 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 addi-
tional 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: the PAUSE,
PAUSEX, and IOWAIT intrinsics will allow the
interrupt. When the interrupt procedure exits, it
reinvokes these intrinsics.*

*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.

There are three types of processes in the system:

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, etc. 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 >>
```

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

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

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.

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 tail of the file; when the file is filled, the next record added causes the block at the head of the file to be deleted and all other blocks to be logically shifted toward the head of the file. Circular files may not be simultaneously accessed by both readers and writers. When the file has been closed by all writers, it may be read; a reader takes records from the circular file one at a time, starting at the head of the file.

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

Creating a circular file is similar to creating a message file. When a user process opens a new file and indicates 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	128	1	8

FEATURES OF INTRINSICS FOR CIRCULAR FILES

Most intrinsics treat circular files the same way they treat regular disc 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 that are omitted 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) - Multiaccess mode. This feature permits processes located in different jobs or sessions to open the same file.

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

        01 - Only intra-job multiaccess allowed; this is the same as specifying the MULTI option in a :FILE command.

        10 - Inter-job multiaccess 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; this determines whether they will be reading the file 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 disc file, but they have different meanings for the readers and writers of a circular file:

USER VALUE	Changed to:	
	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) - Multirecord. 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 disc 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 FDELETE FWRITEDIR FWRITE	FUPDATE FDELETE FWRITEDIR FREAD FREADDIR FREADSEEK FPOINT FSPACE

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

NOTE

Serial disc files are very similar to magnetic tape files. Unless otherwise noted, information in this section applies to serial discs as well as to magnetic tape.

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:

LEADER   BOT   FILE SPACE   EOT   TRAILER
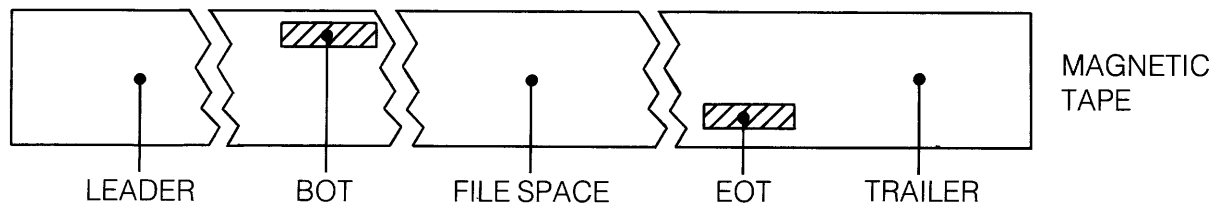
MAGNETIC TAPE

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 5 (%2000) is set and remains on until the drive detects the EOT marker during a rewind or backspace operation. Under normal circumstances, however, it is not necessary to check for EOT during read operations. The responsibility for detecting end of tape and concluding tape operations in an orderly manner belongs to the program 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 at which to break the file. Then several file marks and a trailing tape label would typically be added, the tape rewound, another reel mounted, and the data transfer continued. The program designed to read such a multi-volume file must

expect to find and check for the EOF and label sequence written by the tape's creator. Since the logical end of the tape may be somewhat past the physical EOT marker, the format and conventions used to create the tape are of more importance than determining the location of the EOT.

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):
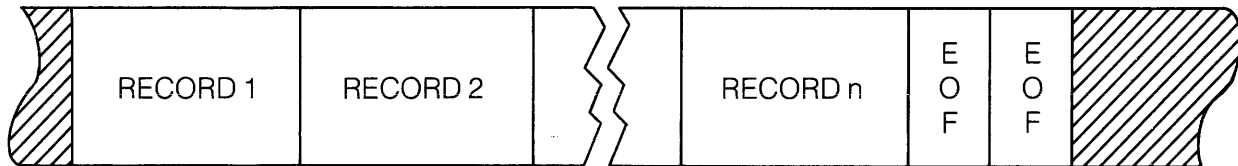
SPACING FILE MARKS

When you space forward to a tape mark (EOF), the tape recording heads have just read the EOF and are positioned beyond it:
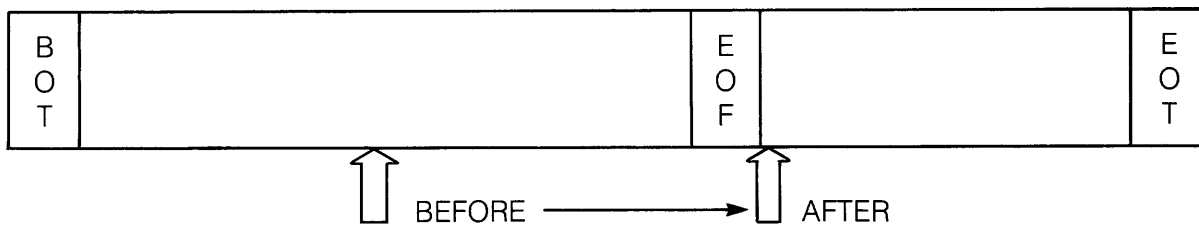
When you space backward to a tape mark (EOF), the mark is recognized as the tape travels in the reverse direction. The tape heads then are left positioned just in front of the EOF that was read:
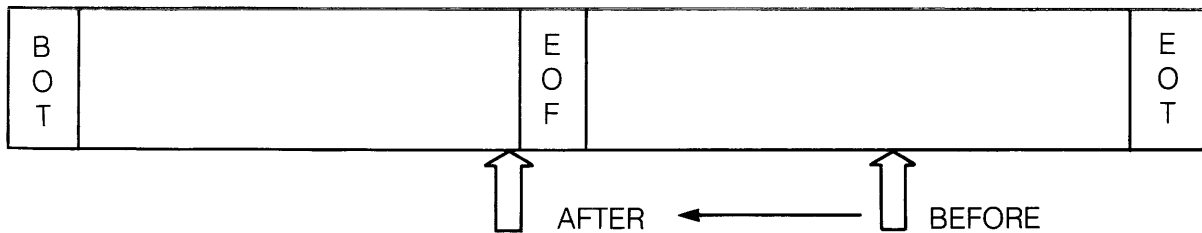
When FREAD has found a logical file mark and returned a condition code of CCG, the EOF mark has been read and the tape heads are positioned immediately following the mark (similar to space forward to tape mark above).

9-3

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.

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

9-4

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 concur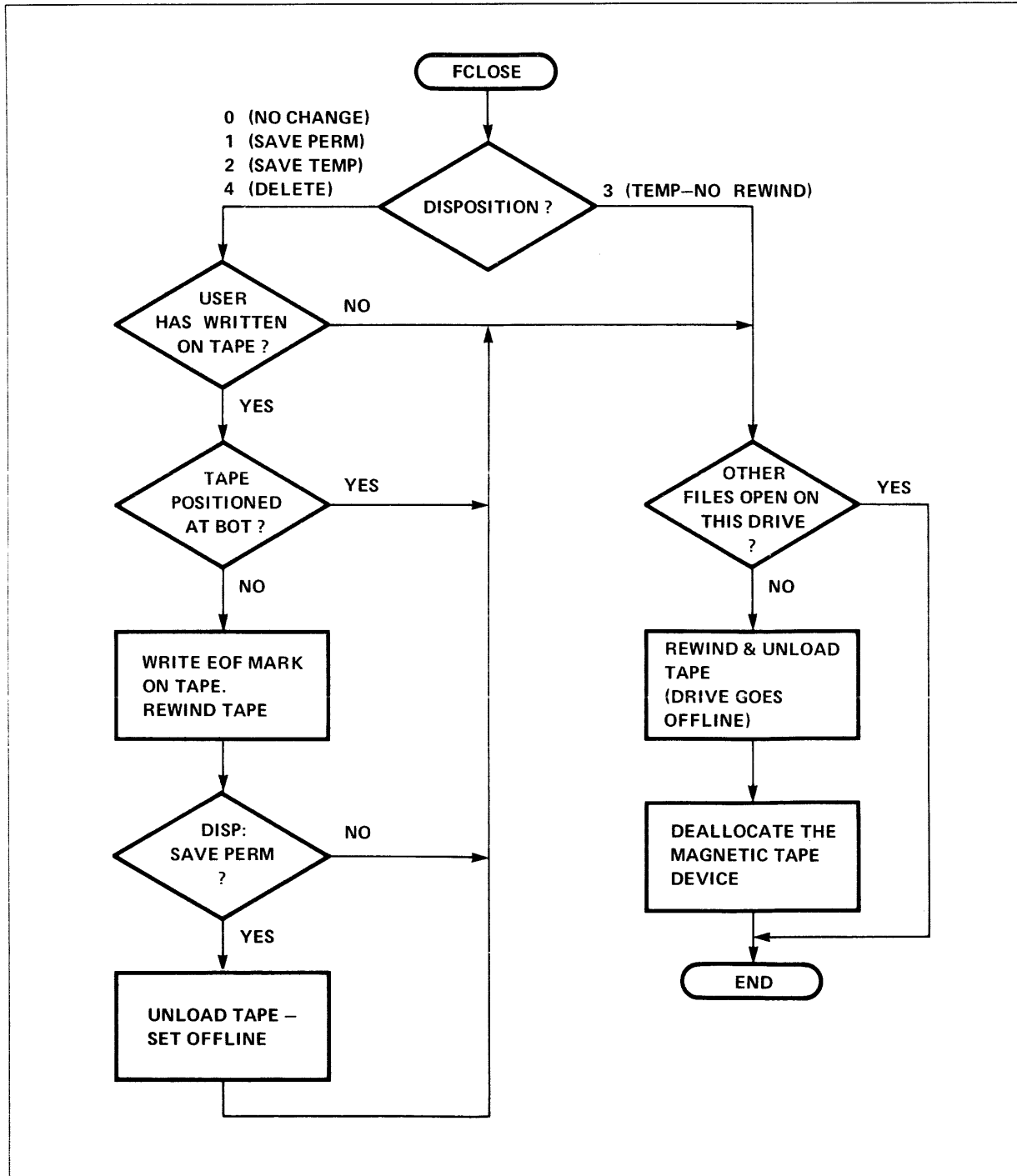rently. The second FOPEN does not require any operator intervention (for example, for device allocation). When FOPEN/FCLOSE calls are arranged in a nested fashion, tape files may be closed without deallocating the physical device, as follows:

```
┌ FOPEN                    allocate tape

    ┌ FOPEN

    └ FCLOSE
                           tape remains allocated
    ┌ FOPEN

    └ FCLOSE

└ FCLOSE                   deallocate tape
```

Such nesting of FOPEN/FCLOSE pairs is required to keep an FCLOSED 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 tape 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:
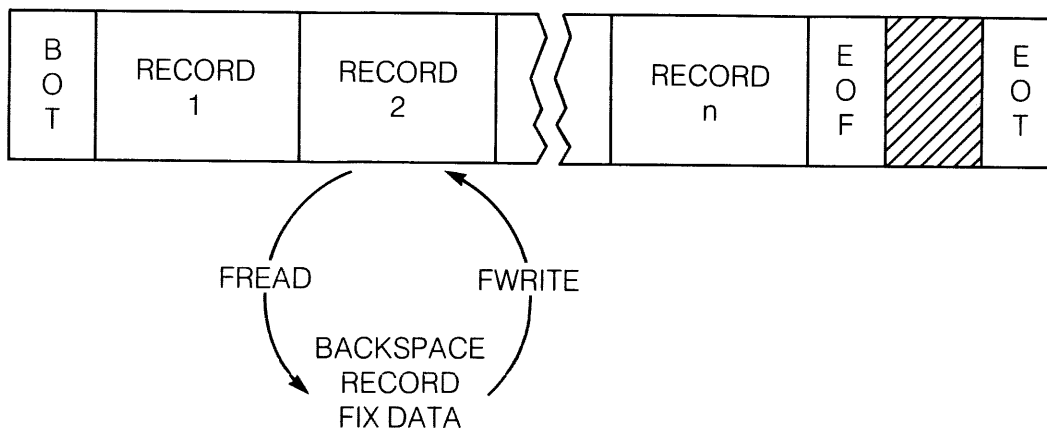
If the replacement differed at all in size from the original record, the result would not simply be an update of the record. A replacement record of greater length than the original record would overwrite (destroy) a portion of the next record on the tape, as shown below.

IRG: INTER-RECORD GAP — THE HARDWARE DEVICE CONTROLLER PLACES A GAP (UNRECORDED TAPE) BETWEEN ADJACENT DATA FIELDS.

REMAINS OF RECORD 3
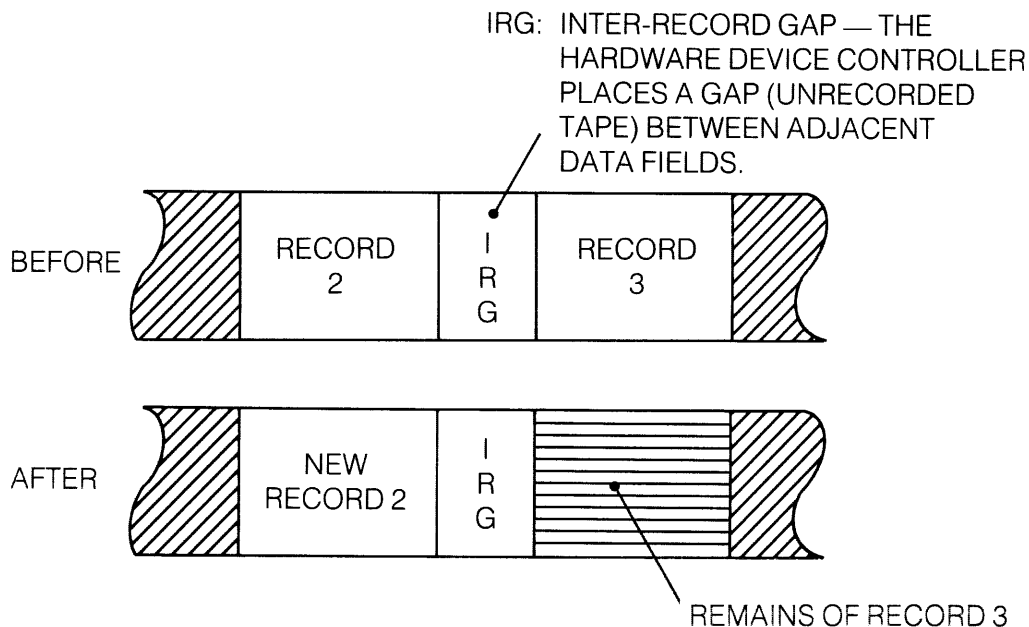
On the other hand, if the length of the replacement record is less than that of the original record, a portion of the original record will still remain on the tape as shown below.
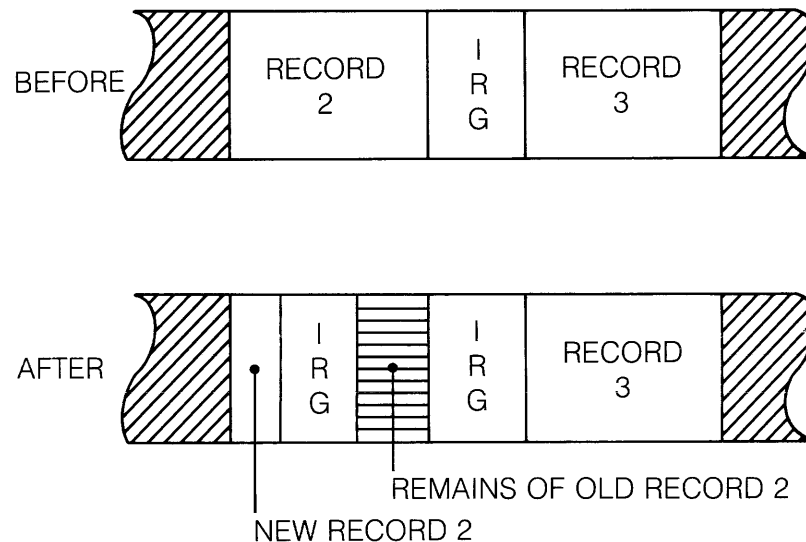
In either of the two cases shown, the partial records remaining would cause magnetic tape read errors and would create problems in subsequent processing of the tape file.

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.05.1   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,FWRITE,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);            <<TAPE FILE 2>>
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
00043000   00104 1         FSPACE(MT,RECD'POSITION);            <<NEXT FILE 1 RECD>>
00044000   00107 1         IF <> THEN FILERROR(MT,8);           <<CHECK FOR ERROR>>
00045000   00113 1         GO COPY'LOOP;                        <<CONTINUE COPYING>>
00047000   00115 1
00048000   00115 1     DONE:
00049000   00115 1         FCONTROL(MT,5,DUMMY);                <<REWIND TAPE>>
00050000   00121 1         IF < THEN FILERROR(MT,9);            <<CHECK FOR ERROR>>
00051000   00125 1
00052000   00125 1         FCLOSE(MT,0,0);                      <<MAG TAPE>>
00053000   00130 1         IF < THEN FILERROR(MT,10);           <<CHECK FOR ERROR>>
00054000   00134 1
00055000   00134 1     END.
     PRIMARY DB STORAGE=%007;   SECONDARY DB STORAGE=%00111
```

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   66 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:
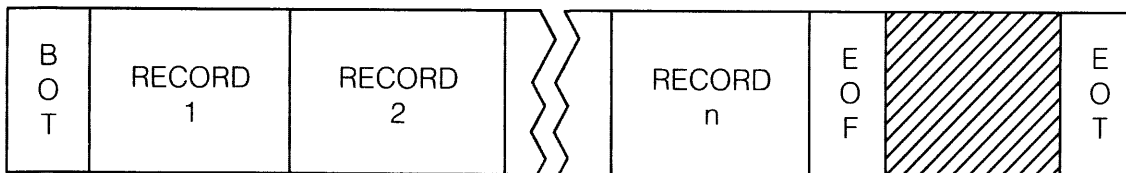
The statement

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:

FILE 1   FILE 2
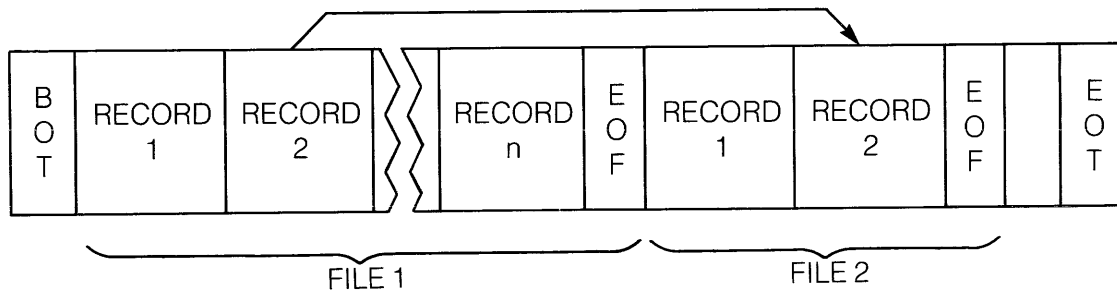
Note that the reverse tape motion after a write creates an EOF mark (see end of FILE 2).

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:

FILE 1   FILE 2
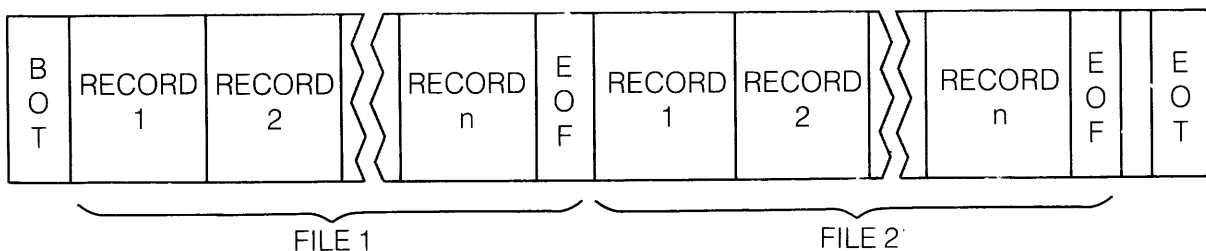
LABELED TAPES

MPE provides a means whereby you can read and write labels on magnetic tape files. Labels on 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 write, you specify the volume set name as 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 reported if a 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.

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 Commands Reference Manual, part number 30000-90009, 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 (0:79): = "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 FNO1,FNO2,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 >>
```

Figure 9-3.   Writing to a Tape File.

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

FNO1: = 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 (FNO1,1);
      END;

   FNO2: = FOPEN (FILID2,% 1004,5,,DEV,LABELID,1);  << OPEN NEW LABELED TAPE
                                                       FILE >>
   IF < THEN                                        << CHECK FOR ERROR >>
      BEGIN
          MOVE MSGBUF: = "CAN'T OPEN TAPE FILE";
          PRINT (MSGBUF,-20,0);
          FILERROR (FNO2,2);
      END;

   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 >>
```

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

9-14

```
    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 (Continued).

The statement

    BYTE ARRAY LABELID (0:79):=".FIL001,ANS,12/31/81,NEXT;";

declares a byte array of 80 bytes and initializes it to

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

which specifies that the tape is to have ANSI-standard labels. Note that the specification begins with a period and ends with a semicolon. This is necessary to distinguish the specification 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

    FNO2: = 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 disc file, reads the contents of the tape file and writes the records to the disc file, closes the tape file, and finally closes the disc file as a permanent file.

```
$CONTROL USLINIT
BEGIN
    BYTE ARRAY FILID1 (0:8): = "TAPEFILE ";
    BYTE ARRAY FILID2 (0:8): = " ";
    BYTE ARRAY LABELID (0:79): = ".FIL001,ANS,12/31/81,,;";
    BYTE ARRAY DEV (0:4): = "TAPE ";

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

    INTEGER FNO1,FNO2,LGTH;
```

Figure 9-4.   Reading a Labeled Magnetic Tape File.

```
INTRINSIC FOPEN,FCLOSE,FREAD,FWRITE,READ,PRINT,PRINT'FILE'INFO,
        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 DISC FILE >>

FNO1: = FOPEN (FILID1, % 1005,5,,DEV,LABELID); << OPEN LABELED TAPE FILE >>
IF < THEN   << CHECK FOR ERROR >>
    BEGIN
        MOVE MSGBUF: = "CAN'T OPEN TAPE FILE";
        PRINT (MSGBUF,-20,0);
        FILERROR (FNO1,1);
    END;

FNO2: = 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 (FNO2,2);
    END;

FREADLABEL (FNO1,INBUF,40);     << READ USER LABEL >>
IF <> THEN FILERROR (FNO1,3);   << CHECK FOR ERROR >>

PRINT (INBUF,40,0);

READ'WRITE'LOOP;
```

Figure 9-4.   Reading a Labeled Magnetic Tape File. (Continued)

```
    LGTH:=FREAD (FNO1,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 (FNO1,4);
       END;
    IF > THEN GO CLOSE1; << CHECK FOR END-OF-FILE >>

    FWRITE (FNO2,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 (FNO2,5);
       END;

GOTO READ'WRITE'LOOP;

CLOSE1:

    FCLOSE (FNO1,1,0); << CLOSE, REWIND, AND UNLOAD TAPE FILE >>
    IF < THEN
                       << CHECK FOR ERROR >>
       BEGIN
          MOVE MSGBUF:="CAN'T CLOSE TAPE FILE";
          PRINT (MSGBUF,-21,0);
          FILERROR (FNO1,6);
       END;

CLOSE2:

    FCLOSE (FNO2,1,0); << 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.   Reading a Labeled Magnetic Tape File. (Continued)

9-18

The statement

    FNO1:=FOPEN (FILID1,% 1005,5,,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.
ASCII/Binary: ASCII. Bit (13:1) = 1.
File Designator: Actual file designator same as formal file designator.
Bits (10:3) = 000. (Default)
Record Format: Fixed length. Bits (8:2) = 00. (Default)
Carriage Control: No carriage control. Bit (7:1) = 0. (Default)
Labeled Tape: Labeled. Bit (6:1) = 1.

aoptions   5, 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 1   Binary

                         5   Octal

The above bit pattern specifies the following access options:

Access Type: Update. Bits (12:4) = 0101.

recsize   Default.

device   TAPE, contained in the byte array DEV.

tape label   Contained in the byte array LABELID (formmsg parameter). LABELID is
                      declared with the value

                      .FIL001,ANS,12/31/81,,;

                      (See statement number 5 in Figure 9-4). The label specification 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 disc file.

The program then reads records from the tape file with the statement

    LGTH: = FREAD (FNO1,INBUF,40);

and writes these records to the disc file with the statement

    FWRITE (FNO2,INBUF,LGTH,0);

When all records in the tape file have been read, both files are closed. The disc 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 defaults to the blocksize, recordsize and file format on the tape label if these parameters are not specified. You can call FGETINFO or FFILEINFO to get these values.

The program that was 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 disc 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 as follows:

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 that was shown in Figure 9-3 opens an existing disc file and a new labeled tape file, reads records from the disc 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.

User-defined labels for labeled tape files differ slightly from user-defined labels for disc 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. User header labels on tapes destined for foreign systems should have the characters "UHL" and a digit as the first four characters of the label, but for MPE, they are arbitrary.

The program that was shown 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 (contained in 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.

40   Specifies the number of words to be read.

DUMPING FILES OFF-LINE

You can obtain a back-up copy of a particular user disc file or set of files by copying it offline onto magnetic tape or serial disc via the :STORE command. If you have standard user capability only, you can dump any file to which you have read-access. If the file has a negative file code, however, you must also have the Privileged Mode Optional Capability to dump this file. If you have Account Manager Capability, you can dump any file in your account, but cannot dump those with negative file codes unless you also have the Privileged Mode Capability. If you have the System Manager or System Supervisor Capability, you can dump any user file in the system. The files are copied in a special format along with all descriptive information (such as account name, group name and lockword), permitting them to be read back into the system later by the :RESTORE command.

The :STORE and :RESTORE commands are used primarily as a back-up for files. However, you can also use them to interchange files between installations if the accounts, groups, and creators of the files to be restored are defined in the destination system. Furthermore, if you specify no destination device in the :RESTORE command, MPE does not guarantee which devices will actually receive the files: if a device of the same type as the original device with sufficient storage space cannot be found, the file is restored to any device that is a member of the device class DISC.

Files currently open for output, input/output, update, or append access cannot be acted upon by a :STORE command. Files currently being stored or·restored cannot be acted upon by a :STORE command. However, files loaded into memory (containing currently running programs), and files open for input only, can be stored, since their contents cannot be altered.

While a file is being dumped, it is locked by MPE so that it cannot be altered or deleted until safely copied to tape or serial disc. If a job/session running a :STORE/:RESTORE function is aborted by yourself or the console operator, those files not yet stored or restored will be unlocked during the processing of the abort.

The flow chart in Figure 9-5 shows the checks performed against a file to ensure its eligibility for dumping.
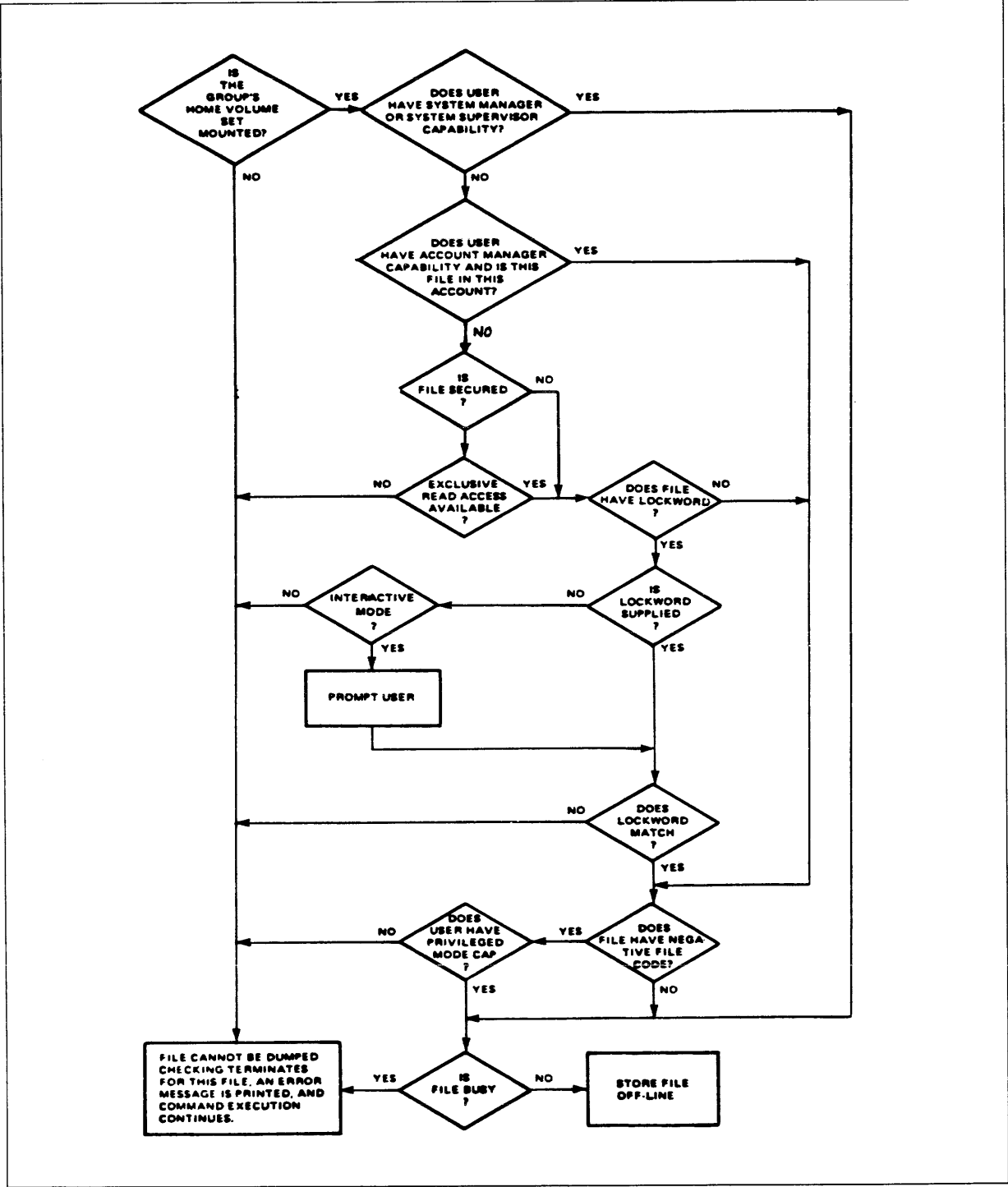
Figure 9-5.   Checks for File Dump Eligibility.

Magnetic Tape Format

Tapes produced by the :STORE command may be labeled or unlabeled. Unlabeled :STORE tapes are compatible with those produced by the :SYSDUMP command used by System Supervisors for general backup of the overall system. (:SYSDUMP tapes are read by the MPE Initiator program to reload the system.) Tapes produced by either :STORE or :SYSDUMP are suitable as input to the :RESTORE command.

NOTE

In general, standard users use :STORE/:RESTORE when they desire to back-up only those files which belong to a particular set of groups or accounts. System Supervisors use :SYSDUMP for daily back-up of the overall system, since it provides a record of the latest accounting information. However, System Supervisors may also use :STORE/:RESTORE to save or load any or all files on the system provided the appropriate account, group, and user structures already exist. Tapes created with :SYSDUMP cannot be labeled.

The general formats of labeled and unlabeled magnetic tapes created by the :STORE command are presented in Figure 9-6. These formats are defined in greater detail in Table 9-1 for labeled tapes, and in Table 9-2 for unlabeled tapes. Both :STORE and :RESTORE support multifile and multi-reel files.

The *tape directory* records are 12 word records with a default *blocksize* of 4096 words. There is one entry for each file on the tape. The entries are ordered the same as the files on the tape (see Table 9-2).

The *recsize* parameter of the :FILE command can be used to change record size. The default record size of each file is 4096 words. The last record may be shorter, but it will be a multiple of 256 words. The beginning of each file contains the system file label known to the file system.
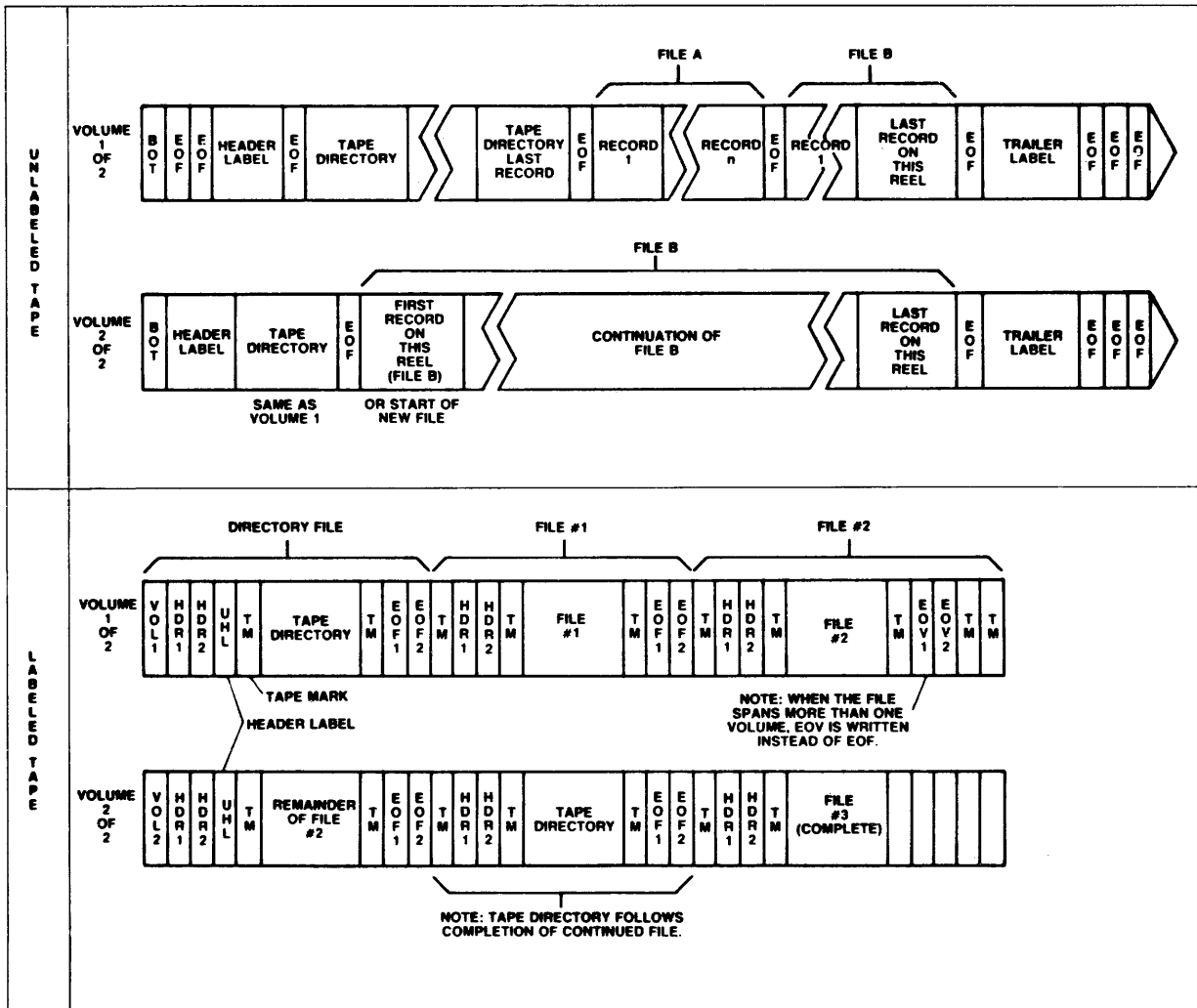
Figure 9-6.   :STORE Tape Formats.

Table 9-1.   Format of Tape Labels written by MPE (ANSI Standard).

File Header Label (80 bytes)		
Position	**Contents**	**Comments**
Bytes 1-4	HDR1	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 22-27	*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 on a reel.
Bytes 36-41	Blanks	Not written by MPE. (Reserved generating data groups in ANSI-standard labels.)
Bytes 42-47	file creation date	Indicates date on which file is written to magnetic tape.
Bytes 48-53	file expiration date	Indicates date after which file can be overwritten.
Byte 54	%230	Indicates that label was created by MPE.
Bytes 55-80	Blanks	Reserved for future use.
Volume Header Label (80 bytes)		
Position	Contents	Comments
Bytes 1-4	VOL1	Indicates volume label (1 specifies 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 written by MPE. (Used 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 9-2.   :STORE Tape Format.

ITEM NO.	ITEM
1	End-of-File (EOF) Mark
2	EOF Mark
3	Header label (40 words), used as follows:

	Words	Contents
	0-13	"STORE/RESTORE LABEL-HP/3000."
	14-15	**MPE III** Identification: (Identifies modified format of :STORE tape).
	16	If true, indicates that first file on volume is continued from previous volume.
	17	Checksum for verifying validity of words 14 and 15.
	18	Index into STORE directory of first file on volume (Could be continuation of a file from previous volume).
	19-21	Reserved by MPE.
	22	Block size in words. "0" means 1024 words.
	23	Reel Number.
	24	Bits (0.7) = last 2 digits of year } Date of creation (7.9) = Julian date
	25	Bits (0.8) = hours (8.8) = minutes
	26	Bits (0.8) = seconds } Time of creation (8.8) = tenth-of-seconds

ITEM NO.	ITEM
4	EOF Mark
5	Tape directory—Consists of 12-word records blocked according to the tape block size specified. (The last block may be shorter.) There is one entry for each file on the tape, and the entries are ordered the same as the files. The 12-word entry is

	Word	Contents
	0-3	File name
	4-7	Group name
	8-11	Account name

ITEM NO.	ITEM
6	EOF Mark
7	First file. The data is blocked according to the block size specified in the store tape :FILE command (The last block may be shorter, but will always be a multiple of 256 words.) For fixed-length and undefined-length record files, only data up to the end-of- file is dumped; intervening zero-length extents are not dumped. For variable-length record files, only allocated extents are dumped.

Table 9-2   :STORE Tape Format (Continued).

ITEM NO.	ITEM
8	EOF Mark
9	Second File
10	EOF Mark
11	Last File
12	EOF Mark
13	Trailer Label (40 words). Identical to header label (Item 3) except that Words 21 and 22 are used as follows **WORD** **USE** 21 = 1 means that preceding file ended with preceding EOF mark 22 = 1 means that entire tape set ends with preceding EOF mark
14	EOF Mark
15	EOF Mark
16	EOF Mark

:STORE tapes may have multiple reels. If end-of tape (EOT) is detected during a write data operation. a file mark is written followed by Items 13 to 16 above. with word 21 of the trailer label set to 1 if this was the last record of the file and 0 otherwise. If EOT is detected on a write file mark operation. Items 13 to 16 are written with word 21 set to 1 and word 22 set to 1 if this is the last file on the tape, and 0 otherwise. Reels subsequent to Reel 1 have the following format

1   Header label

2   EOF mark

3. Remainder of preceding file or next file

4. EOF mark

5   Next file. the rest of the tape is written in the same format as the first reel

Each file on the tape is written in blocks which you may specify in the related :FILE command to be from 256 to 4096 (8192 for serial disc) words long, in increments of 256 words. The default value is 4096 words for non-programmatic calls to :STORE and 1024 words for programmatic calls. The last block may be shorter, but its length will be a multiple of 256 words. The first 128 words of the first block of each file contains the system file label.

Listing Results of the :STORE Command

After the tape is written, MPE prints data showing the results of the :STORE command. By default, this output is sent to the standard list device ($STDLIST). However, you can override this default and transmit the output to another file by issuing a :FILE command equating SYSLIST, the formal designator by which the :STORE command executor references this list file, to another file. For example, if you are located at a terminal, you might transmit this output to a line printer by entering:

    :FILE SYSLIST=MYFILE;DEV=LP

(*Assume the device class LP indicates a line printer.*)

If you omit the SHOW keyword from the :STORE command, only the total number of files actually stored, a list of files not stored, and a count of files not stored, are printed. But if SHOW is included, the listing of files appears in the format shown in Figure 9-7. A sample printout is shown in the lower portion of the figure. In this format, *xxx* is a value denoting the total number of files dumped onto tape; *yyy* denotes the number of files requested that were not dumped. The notations *filename*, *groupname*, and *acctname* under the FILES STORED heading name the individual files dumped, and their groups and accounts, respectively.

The notation *ldn* indicates the logical device number (in decimal) of the device on which the file label and first extent resides, and *addr* is the absolute address (in octal) of the file label. The notation *volume* indicates the volume number of the tape set onto which the file was stored. The notations *filename*, *groupname* and *acctname* under the FILES NOT STORED heading, indicate the individual files not dumped, and their groups and accounts. The notation *fileset#* shows the number of the *fileset* to which the particular file belongs (relative to its position following the :STORE command name). The notation *msg* is a message denoting the reason that the file was not dumped. These errors do not abort the file storing operation, which continues. The messages and their meanings are shown in Table 9-3.

```
FILES STORED = xxx

     FILE            .GROUP          .ACCOUNT        LDN        ADDRESS

     filename1       .groupname1     .acctname1      ldn1       addr1
     filename2       .groupname2     .acctname2      ldn2       addr2
     .
     .
     .
     filenamen       .groupnamen     .acctnamen      ldnn       addrn

FILES NOT STORED = yyy

     FILE            .GROUP          .ACCOUNT        FILESET    REASON

     filename1       .groupname1     .acctname1      fileset    msg
     filename2       .groupname2     .acctname2      fileset    msg
     .
     .
     .
     filenamen       .groupnamen     .acctnamen      fileset    msg


Example

   FILES STORED = 6

     FILE        .GROUP      .ACCOUNT     LDN     ADDRESS      VOLUME

     DATA        .PUB        .SUPPORT     1       %24324       1
     FSMT        .PUB        .SUPPORT     1       %111052      1
     FSMTS       .PUB        .SUPPORT     1       %110775      1
     FTEST       .PUB        .SUPPORT     1       %111237      1
     FTESTJX     .PUB        .SUPPORT     1       %41207       2
     FTESTS      .PUB        .SUPPORT     1       %23603       2

   FILES NOT STORED = 1

     FILE        .GROUP      .ACCOUNT     FILESET              REASON

     K2861445    .PUB        .SUPPORT     1                    BUSY
```

Figure 9-7.   List Output of :STORE Command.

Table 9-3.   :STORE Command Error Messages.

MESSAGE	MEANING
ACCOUNT NOT IN DIRECTORY	Specified account does not exist.
GROUP NOT IN DIRECTORY	Specified group does not exist.
FILE NOT IN DIRECTORY	Specified file does not exist.
BUSY	File is open for output, or is currently being stored or restored.
FILE CODE < 0 AND NO PRIVMODE	You tried to store a file with a negative file code, but do not have Privileged Mode Capability.
LOCKWORD WRONG	The file lockword either was not provided or was specified incorrectly.
READ ACCESS FAILURE	You do not have read access to the specified file.
FILE LABEL ERROR	Due to a problem beyond your control, the file label is not valid.

Examples of backing up files. The following examples illustrate how to make a back-up copy of files. To copy all files in the group GP4M in your log-on account to a tape file named BACKUP, enter the following commands. A listing of files copied and not copied appears on the standard listing device.

```
:FILE BACKUP;DEV=TAPE
:STORE @.GP4M;*BACKUP;SHOW
```

The :STORE command references the formal designator SYSLIST when it lists the files copied and not copied. The output is sent to the standard listing device by default, but by referencing SYSLIST in a :FILE command, you can send the list of files copied and not copied to a file, or to the line printer, as shown below:

```
:FILE SYSLIST;DEV=LP
:FILE BACKUP;DEV=TAPE
:STORE @.GP4M;*BACKUP;SHOW
```

To copy the same set of files to serial disc, enter:

```
:FILE BACKUP;DEV=SDISC
:STORE @.GP4M;*BACKUP;SHOW
```

Explicit or implicit redundant references are permitted among the files you request, but once a file has been locked down for dumping, any subsequent references to it result in the message BUSY even though execution of the :STORE command continues. For instance, suppose the file identified as FN.GN.AN is a member of the fileset referenced by @ in the following command:

    :STORE @,FN.GN.AN;*DUMPTAPE;SHOW

The command is executed successfully, but the fileset# and msg notations under FILES NOT STORED on the listing show:

filename	groupname	acctname	fileset#	msg
FN	GN	AN	2	BUSY

This same file is also noted under FILES STORED. The file is, in fact, actually stored.

Retrieving Dumped Files

A particular file or fileset that has been stored offline by :STORE (or :SYSDUMP) can be copied back onto disc.If you have System Manager of System Supervisor Capability, you can restore any file from a :STORE tape, assuming the account and group to which the file belongs and the user who created the file exist in the system.If you have Account Manager Capability, you can restore any file in your account (but cannot restore those with negative file codes unless you also have Privileged Mode Capability). If you have standard user capability, you can restore any file in your log-on account if you have SAVE access to the group to which the file belongs (but you cannot restore those with negative file codes unless you also have Privileged Mode Capability). If the file to be stored is protected by a lockword, you must supply the lockword in the :RESTORE command; if you are logged on interactively, you will be prompted for the lockword if you fail to supply it.

MPE attempts to restore the files to a device of the same class as the device on which they were originally created. The files are attached to the appropriate groups and accounts with previous account and group names and lockwords all reinstated. The :RESTORE command does not create any new accounts or groups. Any file to be restored is restored only if the account name and group name exist on disc (in the system directory).

If a copy of a file to be restored already exists on disc, you must have write access to the disc file (since it will be purged by :RESTORE) unless you use the KEEP keyword. This keyword specifies that if a file referenced in the :RESTORE command currently exists on disc, the file on disc is retained and the corresponding tape file is not copied into the system.

Files currently open, loaded into memory, or being stored or restored, cannot be acted upon by a :RESTORE command.

The :RESTORE command performs the same checking done by the :STORE command to ensure a file's eligibility for retrieval. If you include the SHOW keyword in the :RESTORE command, MPE prints a listing showing which files were restored. Otherwise, a count of files restored, a list of files not restored, and a count of files not restored, are supplied.

Files can be restored directly from the volume member onto which the file was stored. Thus, if the file was stored on the mth member of a set consisting of n members, the restore operation can start directly from the mth member.

If the location of a file is not known in relation to a specific volume of a multi-volume set, the following strategy can be used to eliminate unnecessary sequential scanning of member volumes for locating specific files:

> Mount the *last* member of the set at the initiation of the restore operation. If this member cannot satisfy the requirement of locating the file, the :RESTORE command will ask that the immediately preceding member of the set be mounted. (The member's number will be displayed in the request message on the console.) This operation of locating the appropriate member will continue until the target member is recognized, at which time the actual restore of the files will begin.

It is strongly recommended that the List Output of the :STORE command be used to precisely determine the location of files stored on multi-volume :STORE sets.

Listing Results of the :RESTORE Command

As with the listing produced by :STORE, the listing output by :RESTORE is transmitted to a file whose formal designator is SYSLIST; if you do not specify otherwise, this file is equated to the standard list device, $STDLIST. An example of a typical :RESTORE operation with SHOW and KEEP options appears in Figure 9-8. The format of the listing is the same as that for the :STORE example shown in Figure 9-6. The notation msg is an error message denoting the reason that the file was not restored. These errors do not abort the file-restoring operation. The messages and their meanings are listed in Table 9-4.

NOTE

> *Tape and serial disc files created by the :SYSDUMP command and the :STORE command are compatible. (:SYSDUMP is discussed in the System Manager/System Supervisor Reference Manual, part number 30000-90014.) Thus, a file dumped via :SYSDUMP can be used as input for the :RESTORE command. However, a :STORE/:RESTORE tape or serial disc pack cannot be used as the first volume of a system initiation input medium during a reload, because the operating system has not been copied to the medium.*

```
FILES RESTORED = 6

    FILE        .GROUP      .ACCOUNT      LDN    ADDRESS

    DATA        .PUB        .SUPPORT       1     % 23463
    FTEST       .PUB        .SUPPORT       1     % 24324
    FTESTJ1     .PUB        .SUPPORT       1     % 23741
    FTESTJOB    .PUB        .SUPPORT       1     % 23753
    FTESTS      .PUB        .SUPPORT       1     % 23765
    PEOF        .PUB        .SUPPORT       1     % 24354

FILES NOT RESTORED = 7

    FILE        .GROUP      .ACCOUNT      FILESET      REASON

    FSMT        .PUB        .SUPPORT         1         ALREADY EXISTS
    FSMTS       .PUB        .SUPPORT         1         ALREADY EXISTS
    FTESTJ2     .PUB        .SUPPORT         1         ALREADY EXISTS
    FTESTJ3     .PUB        .SUPPORT         1         ALREADY EXISTS
    FTESTJX     .PUB        .SUPPORT         1         ALREADY EXISTS
    JUNKJOB     .PUB        .SUPPORT         1         ALREADY EXISTS
    PEOFS       .PUB        .SUPPORT         1         ALREADY EXISTS
```

Figure 9-8.   List Output of :RESTORE with SHOW and KEEP.

Table 9-4.   :RESTORE Command Error Messages.

MESSAGE	MEANING
ACCOUNT DIFFERENT FROM LOGON	The file's account name is different from the name of your log-on account. Users do not have save-access to groups outside their log-on accounts.
ACCOUNT DISC SPACE EXCEEDED	The account's disc space limit would be exceeded by restoring this file.
ALREADY EXISTS	A copy of the file specified already exists on disc, and KEEP was also specified. The file was not replaced.
BUSY	The disc file is open, loaded, or being stored or restored at present.

Table 9-4.   :RESTORE Command Error Messages. (Continued)

MESSAGE	MEANING
CATASTROPHIC ERROR	A catastrophic error occurred while the system was restoring either this file or one previous to it on the tape, and the :RESTORE command was aborted. This message may result from one of the following:
	Command Syntax error.
	Disc input/output error (in system).
	File directory error.
	File system error on the tape file (TAPE), list file (LIST), or any of the three temporary files (GOOD, ERROR, and CANDIDAT) used by the :RESTORE command executor.
	Improper tape; the tape used for input was not written in :STORE/:RESTORE format.
	No continuation reel; the computer operator could not find a continuation reel for a multi-reel tape set.
	Device reference error; the specification for the *device* parameter is illegal, or the device requested is not available.
	Tape read error in a sensitive part of the tape, which makes it impossible to continue. (Most tape errors are merely skipped, omitting the affected file.)
CREATOR NOT IN DIRECTORY	The creator of the file is not defined in the system.
DISC FILE CODE < 0 AND NO PRIV MODE	One of the files (on disc) to be replaced has a negative file code, and you do not have Privileged Mode Capability.
DISC FILE LOCKWORD WRONG	The disc file has a lockword that does not match the lockword for the file on tape.
FILE LABEL ERROR	Due to a problem beyond your control, the file label is not valid.

Table 9-4. :RESTORE Command Error Messages. (Continued)

MESSAGE	MEANING
GROUP DISC SPACE EXCEEDED	The group's disc space limit would be exceeded by restoring this file.
GROUP NOT IN DIRECTORY	The group specified does not exist in the system.
NOT ON TAPE	The file specified is not on the tape.
OUT OF DISC SPACE	There is insufficient disc space to restore this file.
SAVE ACCESS FAILURE	You do not have save-access to the group to which the file belongs.
TAPE FILE CODE < 0 AND NO PRIV MODE	One of the files on tape to be restored has a negative file code, and you do not have Privileged Mode Capability.
TAPE FILE LOCKWORD WRONG	The tape file has a lockword that you did not supply or did not specify correctly.
TAPE READ ERROR	A tape read error has occurred on a block other than that containing the file label.
WRITE ACCESS FAILURE	You do not have write-access to the copy of the file on disc.

Examples of restoring files. The following examples show how to retrieve files using :RESTORE. To retrieve from the file named BACKUP all files formerly belonging to your log-on group, enter:

    :FILE BACKUP;DEV=TAPE
    :RESTORE *BACKUP;@;KEEP;DEV=MHDISC;SHOW

To have the list of restored files printed on a line printer, enter:

    :FILE SYSLIST;DEV=LP
    :FILE BACKUP;DEV=TAPE
    :RESTORE *BACKUP;@;KEEP;DEV=MHDISC;SHOW

If a file satisfying the "@" specification already exists in the system, it is not restored.

RECORDS - Always begin and end on word boundaries (odd byte length records padded out to a word boundary).

Record Formats

Fixed

All records a fixed length; blocks contain a fixed number of records.

Record length and blocking factor known to file system.

Records consist of data only.

Variable

Record length varies; blocks contain a variable number of records.

File attribute defined to file system is Maximum Record Length = (Record Length + 1) $\times$ 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

Block length set to record length defined to file system.

Blocking factor assumed to be 1.

Blocking and deblocking of records is the user's responsibility.

Records shorter than block length have their last word of data repeated to end of block.

Buffering

```
: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 the :FILE and :BUILD Commands

```
                                       [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:

−400		IMAGE root file.
−401		IMAGE data set.
−402		IMAGE file for DS information.
1024	USL	USL file.
1025	BASD	BASIC data file.
1026	BASP	BASIC program file.
1027	BASFP	BASIC fast program file.
1028	RL	Relocatable Library.
1029	PROG	Program file.
1031	SL	Segmented Library.
1035	VFORM	V/3000 forms file.
1036	VFAST	V/3000 fast forms file.
1037	VREF	V/3000 reformat file.
1040	XLSAV	Cross Loader ASCII file (SAVE).
1041	XLBIN	Cross Loader relocated binary file.
1042	XLDSP	Cross Loader ASCII file (DISPLAY).
1050	EDITQ	Edit KEEPQ file (non-COBOL).
1051	EDTCQ	Edit KEEPQ file (COBOL).
1052	EDTCT	Edit TEXT file (COBOL).
1058		TDP/3000 work file.
1060	RJEPN	RJE punch file.
1070	QPROC	QUERY procedure file.
1071		QUERY work file.
1072		QUERY work file.

```
1080    KSAMK    KSAM key file.
1083    GRAPH    GRAPH specification file.
1084    SD       Self-describing file.
1090    LOG      User Logging file.
1110    PCELL    IDS/3000 character set file.
1111    PFORM    IDS/3000 form file.
1112    P2680    IFS/3000 environment file.
1130    OPTLF    On-line performance tool.
```

[;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]

> :BUILD command can only create a file in the Permanent or Temporary domain. Default is Permanent.

Referencing Disc File Domains

```
          [,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 assumes same disposition as at open.

:FILE Back-Reference

: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

```
        [;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

```
        IN
        OUT
        INOUT
:FILE . . . [;ACC = OUTKEEP ]
        APPEND
        UPDATE
```

IN: read only.

OUT: write only. Original contents of file overwritten. File cannot be read.

INOUT: any operation but update allowed. (You can still read then write 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 original extent.

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

MULTI: requires Process Handling capability.

MR: allows multiple block access.

NOWAIT: requires Privileged Mode capability.

User Types

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.

FOPTIONs for Use with FOPEN

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≡$STDLIST	1≡ASCII	01≡Old Permanent File
		01	0≡RIO				10≡Unde-fined	010≡$NEWPASS		10≡Old Temporary File
		10	0≡CIR					011≡$OLDPASS		11≡Old Perm. or Temp. File
		11	0≡MSG					100≡$STDIN		
								101≡$STDINX		
								110≡$NULL		

AOPTIONs for Use with FOPEN

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

MPE Defaults and Device-Dependent Restrictions

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/OUTPUT 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

Relative I/O Block Format

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 $= \dfrac{F}{16}$

$F =$ 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 $= R \bmod 16$

bit $= 0$: inactive record
    $= 1$: active record

You can use certain file system intrinsics to obtain information on the status of your disc files. Information is available about:

Actual file characteristics. These 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. This 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 the logical and physical record transfer count.

Error information. Here you may discover the last error for your file, and/or the last FOPEN error.

OBTAINING STATUS INFORMATION

The same status information may be obtained via different intrinsic calls. The FGETINFO intrinsic will return actual file characteristics and current file information, the FCHECK intrinsic will return error information, and the PRINT'FILE'INFO intrinsic will list details of all three categories. PRINT'FILE'INFO will format information and output it to the list device for your job/session; FGETINFO and FCHECK will return unformatted information directly to your calling program.

PRINT'FILE'INFO.

The PRINT'FILE'INFO 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 or a short file information display for files that are not open.

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          !
+-------------------------------------------------+
```

FILE IS OPEN

- File number represents a currently open file.
- Error indicates last error on file.
- Full display condensed when file is not open.

Figure B-1.   FILE INFORMATION DISPLAY - Full.

```
+-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       !
+--------------------------------------------------+
```

FILE 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.

Figure B-2.   FILE INFORMATION DISPLAY - Short.

Data in a FILE INFORMATION DISPLAY. Sections of the full File Information Display yield different types of information, as indicated in Figure B-3:
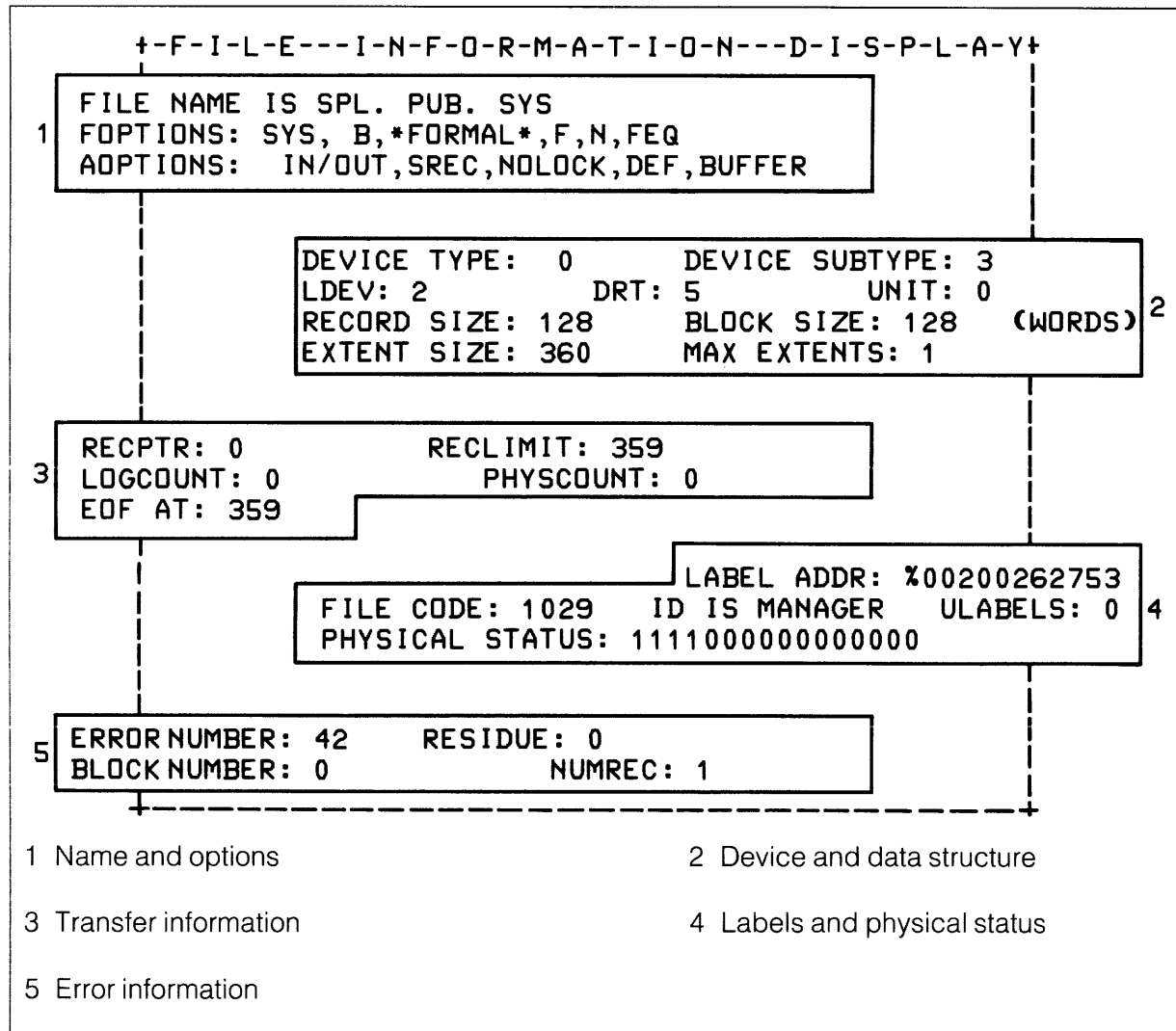
```
      +-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
                                                       |
     +----------------------------------------+        |
     | FILE NAME IS SPL. PUB. SYS             |        |
   1 | FOPTIONS: SYS, B,*FORMAL*,F,N,FEQ      |        |
     | AOPTIONS:  IN/OUT,SREC,NOLOCK,DEF,BUFFER|       |
     +----------------------------------------+        |
          |                                            |
          |    +---------------------------------------+------+
          |    | DEVICE TYPE:   0      DEVICE SUBTYPE: 3      |
          |    | LDEV: 2        DRT: 5         UNIT: 0        | 2
          |    | RECORD SIZE: 128      BLOCK SIZE: 128  (WORDS)|
          |    | EXTENT SIZE: 360      MAX EXTENTS: 1         |
          |    +---------------------------------------+------+
          |                                            |
     +----------------------------------+              |
     | RECPTR: 0          RECLIMIT: 359 |              |
   3 | LOGCOUNT: 0        PHYSCOUNT: 0  |              |
     | EOF AT: 359  +-------------------+              |
     +-------------+--------------------+              |
          |                +------------LABEL ADDR: %00200262753 |
          |                | FILE CODE: 1029   ID IS MANAGER   ULABELS: 0 | 4
          |                | PHYSICAL STATUS: 1111000000000000 |
          |                +---------------------------------+ |
          |                                                   |
     +----------------------------------+                     |
   5 | ERROR NUMBER: 42    RESIDUE: 0   |                     |
     | BLOCK NUMBER: 0         NUMREC: 1|                     |
     +----------------------------------+                     |
      +--------------------------------------------------------+

   1 Name and options              2 Device and data structure

   3 Transfer information          4 Labels and physical status

   5 Error information
```

Figure B-3. Data in a FILE INFORMATION DISPLAY.

The fields in a file information display are described in Tables B-1 through B-5.

Table B-1. Name and Options in a File Information Display.

```
FILE NAME IS SPL.PUB.SYS
FOPTIONS: SYS, B, *FORMAL*, 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

FOPTIONS Keywords

Domain	ASCII/ Binary	Default Designator	Record Format	Carriage Control	Disallow :FILE
NEW SYS JOB ALL	A B	*FORMAL* $STDLIST $NEWPASS $OLDPASS $STDIN $STDINX $NULL	F V U ?	N C	FEQ DEQ

AOPTIONS Keywords

Access Type	Multi-Record	Dynamic Locking	Exclusive Access	Inhibit Buffering
INPUT OUTPUT OUTKEEP APPEND IN/OUT UPDATE	SREC MREC	NOLOCK LOCK	DEF EXC SEA* SHR	BUFFER NOBUFF
Note: Multi-access, NOWAIT fields not represented.				

*Semi-exclusive access (SEMI).

Table B-2.   Device and Data Structure in a File Information Display.

```
        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
```

● Device Type, Subtype   Hardware Information
  LDEV, DRT, UNIT   (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.

Table B-3.   Transfer Information 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 (disc) since FOPEN.

● EOF: 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. If NOBUF, LOGCOUNT = PHYSCOUNT. PHYSCOUNT updated only on completion of I/O transfer.

Table B-4.   Labels and Physical Status in a File Information Display.

```
                                    LABEL ADDR: %00200262753
             FILE CODE: 1029    ID IS MANAGER    ULABELS: 0
             PHYSICAL STATUS: 1111000000000000
```

● LABEL ADDR: Sector address and Idev number for file label. First three digits for Idev; next 8 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 disc at time of last interrupt. (Meaningless for disc in multiprogramming environment.)

Table B-5.   Error Information in a File Information Display.

```
     ERROR NUMBER: 42              RESIDUE:  0
     BLOCK NUMBER: 0                       NUMREC: 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.

FGETINFO and FCHECK

Much of the status information obtainable through the PRINT'FILE'INFO intrinsic can be discovered by using the FGETINFO and FCHECK intrinsics. While PRINT'FILE'INFO prints a file information display, FGETINFO and FCHECK return status information directly to your program through their parameters.

The information returned by FGETINFO and FCHECK that corresponds to PRINT'FILE'INFO information is shown in Figure B-4.

```
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: 111000000000000                  |
       |  ┌──────────────────────────────────────────────┐ |
       |  │  ERROR NUMBER: 42     RESIDUE: 0               │ |  ┌┐
       |  │  BLOCK NUMBER: 0            NUMREC: 1          │ | ⇐││
       |  └──────────────────────────────────────────────┘ |   ││
    +─────────────────────────────────────────────────────+   ┘┘
                                                           FCHECK
```

Note: Physical status (not useful for disc) is obtained through FCONTROL.

Figure B-4. 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 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 errorcode returned is the error which occurred on the most recent intrinsic call or the last FOPEN error.

The relationship between PRINT'FILE'INFO fields and FGETINFO and FCHECK parameters is outlined in Table B-6.

Table B-6.   Parameter/Field Relationships.

FGETINFO/PRINT'FILE'INFO

FGETINFO Parameters	PRINT'FILE'INFO Fields	FGETINFO Parameters	PRINT'FILE'INFO Fields
FILENAME	FILE NAME	LOGCOUNT	LOGCOUNT
FOPTIONS	FOPTIONS	PHYSCOUNT	PHYSCOUNT
AOPTIONS	AOPTIONS	BLKSIZE	BLOCK SIZE
		EXTSIZE	EXTENT SIZE
RECSIZE	RECORD SIZE	NUMEXTENTS	MAX EXTENTS
DEVTYPE	DEVICE TYPE, SUBTYPE		
LDNUM	LDEV	USERLABELS	ULABELS
HDADDR	DRT, UNIT	CREATORID	ID IS
		LABADDR	LABEL ADDR
FILECODE	FILE CODE		
RECPTR	RECPTR		
EOF	EOF AT		
FLIMIT	RECLIMIT		

FCHECK/PRINT'FILE'INFO

FCHECK PARAMETERS	PRINT'FILE'INFO FIELDS
ERRORCODE	ERROR NUMBER
TLOG	RESIDUE
BLKNUM	BLOCK NUMBER
NUMRECS	NUMREC

TERMINAL CHARACTERISTICS

Terminals and character printers, such as the HP 2631B, 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, part number 32193-90002.

This section 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 ENQ's 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 section. 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 section.

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 on line. (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.

For more information about the FCONTROL intrinsic, see the MPE Intrinsics Reference Manual, part number 30000-90010.

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 terminals and classes of terminals shown in Table C-2 are supported by MPE. Terminals equipped with the automatic linefeed feature (operator selectable) must be operated with this feature OFF.

Table C-2. Point-to-Point Terminal Types.

Device Class or Terminal Model	Type	Supported on: II/III	30/33	44	Response to Backspace (H^C) (see note F.)	Character Width	X-ON/X-OFF	ENQ/ACK	Delay Characters g	Block Mode Available	Comments
ASR 33	0	x			\(%134)	7/8	x		x		See Note a
ASR 37	1	x			LF	7/8	x		x		
ASR 35	2	x			\(%134)	7/8	x		x		
Execuport	3	x			LF	7/8	x		x		
Data Point	4	x	x	x	Y^C	7/8	x		x		See Note a
Memorex 1240	5	x			LF	7/8	x		x		
General Non-HP Hardcopy	6	x	x	x	LF	7/8	x		x		See Note b
General Non-HP CRT	9	x	x	x	none	7/8	x		x		See Note c
General HP CRT	10	x	x	x	none	7/8	x	x		x	
Special HP d	11	x	x		none	7/8	x	x		x	
Special HP CRT (HP 2645K)	12	x	x	x	none	8	x	x		x	8 data bits; no parity
Packet Network Interface	13	x	x	x	none	7/8	x				Echo initially off.
Special HP Hardcopy (HP 2635)	15	x	x	x	LF	8	x	x			8 data bits; no parity
General HP Hardcopy (HP 2635)	16	x	x	x	LF	7/8	x	x			
General Non-HP	18	x	x	x	none	7/8	x				See Note e
HP 2631B	19	x	x	x	none	7/8	x				Output only

a   Device does not respond to Form Feed (%14). MPE-inserted Form Feeds replaced by Line Feeds.

b   Originally intended for GE Terminets.

c   Originally intended for Mini Bee; Series II/III strips the character pairs (ESC)A →(ESC)E, (ESC)H, (ESC)J, (ESC)K from input stream.

d   This type allows mixed character and "Block Line" mode. Its use is strongly discouraged.

e   No DCI Read Trigger.

f   This special response is made only for the first backspace character following a data character.

g   Delay characters are output following the carriage return and Line Feed characters to allow the terminal to move the paper or print head.

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-3.

Table C-3. Parity Sensing with the ATC, ADCC and ATP.

	Parity 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 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 III/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 to 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).

C-7

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, even 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 follows:

	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 are 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 checked, 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 timer. 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 file number of the terminal.

controlcode   *Integer by value (required)*. The integer 22.

inputtime   *Logical (required)*. A word to which is returned the input time (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,TIMEOUT);

is used to set a time-out interval of 10 seconds (see statement number 5 in the program). If there is no response to the FREAD intrinsic call (statement number 33) within 10 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 10 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.05.1  SPL/3000  TUE, NOV 25, 1975,  3:53 PM

00001000  00000 0   $CONTROL USLINIT
00002000  00000 0   BEGIN
00003000  00000 1     BYTE ARRAY INNAME(0:5):="INPUT ";
00004000  00004 1     BYTE ARRAY OUTNAME(0:6):="OUTPUT ";
00005000  00005 1     INTEGER IN,OUT,LGTH,DUMMY,TIME,TIMEOUT:=10;
00006000  00005 1     ARRAY BUFR(0:3):="TYPE X",0;
00007000  00004 1     BYTE ARRAY CBUF(*)=BUFR;
00008000  00004 1     ARRAY INSTRUCTIONS(0:34):="REACTION TIMER: ",%6412,
00009000  00011 1       "TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN. ";
00010000  00043 1     ARRAY MSG(0:24):="TRY AGAIN? (Y/N)","WRONG CHARACTER.",
00011000  00020 1       %6412,"YOU'RE TOO SLOW!";
00012000  00031 1     ARRAY RESPONSE(0:16):="REACTION TIME:        MILLISECONDS";
00013000  00021 1     BYTE ARRAY CRESP(*)=RESPONSE;
00014000  00021 1
00015000  00021 1     INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,ASCII,TIMER,QUIT;
00016000  00021 1
00017000  00021 1     <<END OF DECLARATIONS>>
00018000  00021 1
00019000  00021 1         IN:=FOPEN(INNAME,%45);            <<&STDIN>>
00020000  00007 1         IF < THEN QUIT(1);                <<CHECK FOR ERROR>>
00021000  00012 1         OUT:=FOPEN(OUTNAME,%414,%1);      <<$STDLIST>>
00022000  00022 1         IF < THEN QUIT(2);                <<CHECK FOR ERROR>>
00023000  00025 1         FWRITE(OUT,INSTRUCTIONS,35,0);    <<USER 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),9,0);      <<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 GO 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. WARNINGS=000
   PROCESSOR TIME=0:00:03;     ELAPSED TIME=0:00:10
```

Figure C-1.   Using the FCONTROL Intrinsic to Enable and Read the Terminal Input Timer

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		%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

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 subsystem 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 Intrinsics Reference Manual, part number 30000-90010.

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 *n* H^c's.)
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, LF's 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).

Table C-4. Special Characters. (Continued)

Character	Meaning
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.
S^c (DC3, X-OFF)	Suspends the write operation during output. Output may be resumed with Q^c.
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);	Places 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, a colon, or a semicolon) is read as a 2-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. Use of timers on block mode reads is strongly encouraged, 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.

When the echo facility is *off*, input read from the terminal is not echoed to the terminal screen. If the terminal is in half-duplex mode, the input is copied by the terminal itself, and appears as normal printed lines. Bear in mind that the only way printing can be suppressed is with the echo facility off and the terminal in 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

(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 Intrinsics Reference Manual, part number 30000-90009).

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 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.
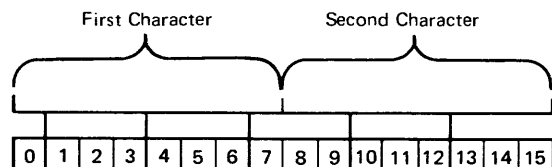
As an example, the following intrinsic call could be used to enable tape mode:

```
    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 only disables 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 end-of-file pointer in the disc file to the proper position. If you are reading more than one tape, you should specify, in the FOPEN intrinsic call that opens the disc file, the append-only access type and a variable-length record format before the first PTAPE intrinsic call. In addition, you should set the disc file's end-of-file pointer to zero, if necessary, before issuing the first PTAPE intrinsic call.

A PTAPE intrinsic call such as

```
PTAPE (TERMFN,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 disc file whose file number is specified by DISCFL.

To inhibit carriage return and linefeed after FREAD, use the FSETMODE intrinsic (see the MPE Intrinsics Reference Manual, part number 30000-90010).

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, or embedding the character pair DC2 CR 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.

ASCII CHARACTER SET

ASCII Character	First Character Octal Equivalent	Second Character 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
O	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
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

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
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
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	000052
+	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
Δ	057000	000136
—	057400	000137
`	060000	000140
{	075400	000173
		076000	000174
}	076400	000175
~	077000	000176
DEL	077400	000177

First Character | Second Character

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

Whenever a disc 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 Commands Reference Manual*.

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.

Words		Contents
29	(Bits 0:8)	Number of user labels written.
	(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		Foptions 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.

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 3 or the byte count is less than 6, 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 occurrence of data beyond an end-of-file and to prevent reading off the end of the reel.

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 written on tape 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 MPE labels.
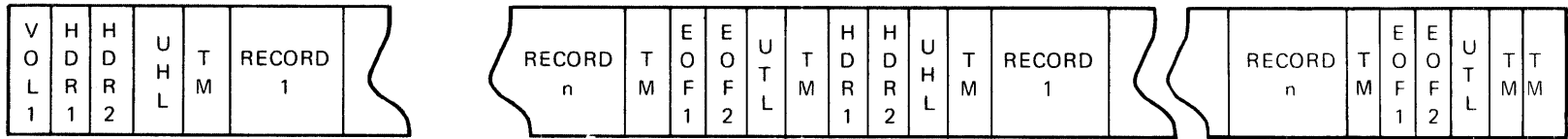
MULTIPLE FILES ON A SINGLE VOLUME

Note: When the file spans
more than one volume, EOV
is written instead of EOF.

VOLUME 1 OF 3 FILE A FILE B

VOLUME 2 OF 3 FILE B

VOLUME 3 OF 3 FILE B FILE C
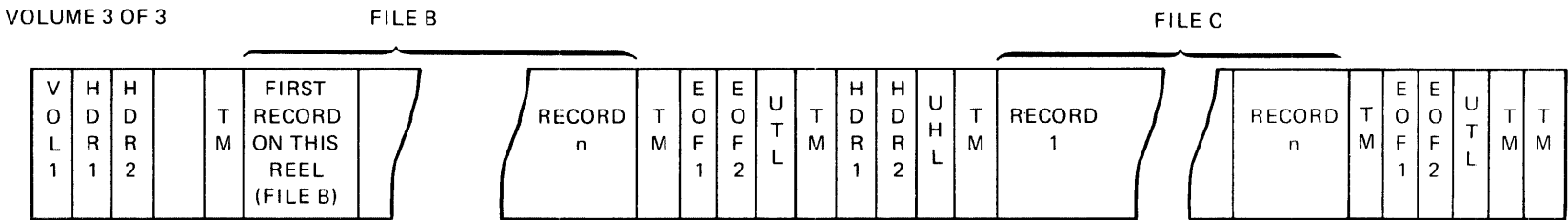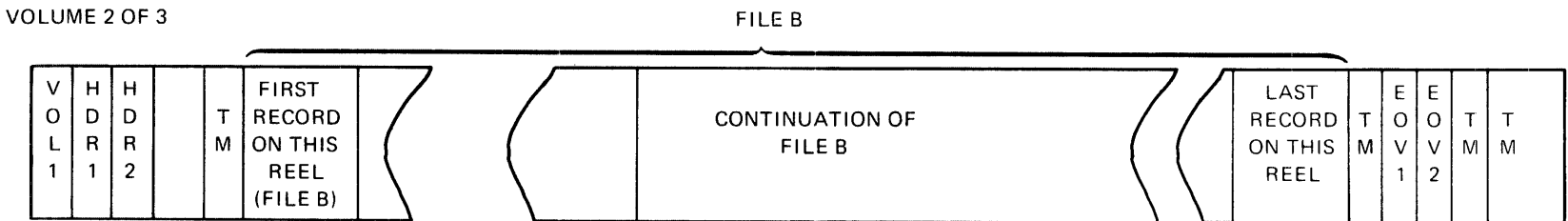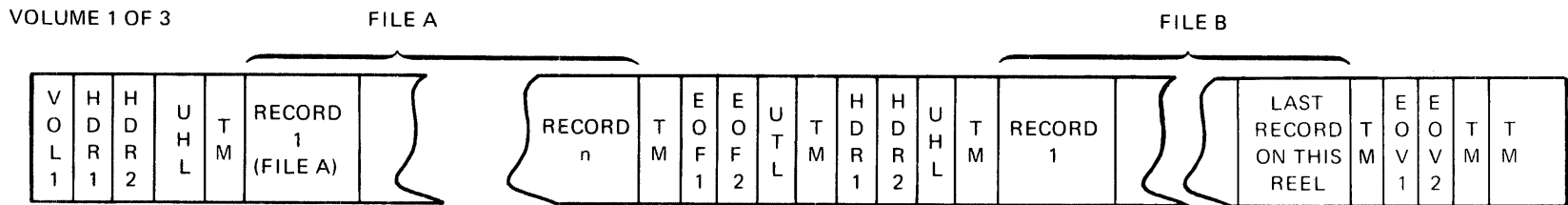
MULTIPLE FILES ON MULTIPLE VOLUMES

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 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 written by MPE. (Used 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.

FILE HEADER LABEL (80 BYTES)

POSITION	CONTENTS	COMMENTS
Bytes 1-4	HDR1	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 22-27	*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 on a reel.
Bytes 36-41	Blanks	Not written by MPE. (Reserved for generating data groups in ANSI-standard labels.)
Bytes 42-47	file creation date	Indicates date on which file is written to magnetic tape.
Bytes 48-53	file expiration date	Indicates date after which file can be overwritten.
Byte 54	%230	Indicates that file was created by MPE and the file has a lockword.
Bytes 55-80	Blanks	Reserved for future use.

INDEX

INDEX

INDEX

READER COMMENT SHEET

MPE File System
Reference Manual

30000-90236   **Feb 1982**

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

Is this manual technically accurate?	**Yes** ☐ **No** ☐	(If no, explain under Comments, below.)
Are the concepts and wording easy to understand?	**Yes** ☐ **No** ☐	(If no, explain under Comments, below.)
Is the format of this manual convenient in size, arrangement, and readability?	**Yes** ☐ **No** ☐	(If no, explain or suggest improvements under Comments, below.)

Comments:

DATE: _____

FROM:

Name   _____

Company   _____

Address   _____

_____

_____

FOLD   FOLD

‖ ‖‖

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS   PERMIT NO. 1070   CUPERTINO,CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
Computer Systems Division
19447 Pruneridge Avenue
Cupertino, California 95014

FOLD   FOLD

HEWLETT
PACKARD